

# COMPUTER VISION

Algorithms and Applications in Computer Vision - 046746

## ABSTRACT

The course focuses on fundamental problems in computer vision describing practical solutions including state-of-the-art deep learning approaches. The topics covered in the course range from semantic tasks like classification and segmentation to structure from motion (SfM), 3D reconstruction, and stereo imaging. The course includes Python homework exercises aimed at enhancing the understanding of the various techniques as well as exposing students to actual implementation details.

Alexander Yorov

Winter – 2022/23

## TABLE OF CONTENTS

Course Agenda .....	5
Ethics in Computer Vision .....	8
How to use computer vision ethically .....	9
Image Understanding .....	10
Image Representation .....	10
Data Sets .....	15
PASCAL VOC .....	15
CIFAR-10 & CIFAR-100 .....	15
ImageNet .....	16
MNIST .....	16
Kinetics-700 .....	16
Convolutional Neural Networks .....	18
Statistical Learning Approach .....	18
History .....	20
Intro to CNNs .....	21
CNN .....	26
Training .....	34
Why it works .....	44
Transfer Learning .....	45
Classical Supervised Learning .....	45
Transfer Learning .....	46
Why Does Transfer Learning Work? .....	49
Recognition .....	50
Object Recognition Tasks .....	50
What matters in recognition? .....	53
What's still hard? .....	53
Image Segmentation .....	54
Semantic Segmentation .....	54
Instance Segmentation .....	54
Segmentation as Clustering .....	55
Graph Based Segmentation .....	59

Evaluation Metrics .....	65
Tools .....	74
LeNet-5 (1998) .....	75
AlexNet (2012) .....	75
VGG Network .....	76
ResNet .....	77
GoogLeNet .....	83
MobileNet .....	85
R-CNN .....	94
Fast R-CNN .....	99
Single Shot Multibox Detector (SSD) .....	103
You Only Look Once (Yolo) .....	106
Feature Pyramid Network (FPN) .....	110
RetinaNet .....	114
SegNet .....	118
Which algorithm to use? .....	119
Pyramid Scene Parsing Network (PSPNet) .....	119
Mask R-CNN .....	121
DeepLab .....	123
DispNet .....	129
GC-Net .....	129
Stereo Mixture Density Networks (SMD-Net) .....	130
Generic Object Tracking Using Regression Networks (GOTURN) .....	131
Multi-Domain Convolutional Neural Network Tracker (MDNet) .....	133
Deep Simple Online and Realtime Tracking (DeepSORT) .....	137
Spatial CNN (SCNN) .....	139
Simple Framework for Contrastive Learning of Visual Representations (SimCLR) .....	140
Momentum Contrast (MoCo) .....	144
Barlow Twins .....	146
Cameras .....	153
Pinhole Camera (Camera Obscura) .....	153
Mathematical Notations .....	157

Direct Linear Transform.....	163
Camera Calibration (Pose Estimation) .....	165
Stereo 3D .....	184
Uncalibrated Stereo (Triangulation) .....	191
Features .....	213
Local Features .....	213
Edge detection .....	213
Corner Detection .....	220
Boundary Detection .....	224
SIFT Detector.....	237
Descriptors.....	244
Image Stitching .....	245
2x2 Image Transformations .....	245
3x3 Image Transformations .....	248
Computing Homography .....	255
Dealing with Outliers – RANSAC.....	258
Warping and Blending Images .....	260
Structure from motion (SfM) .....	265
From 3D to 2D: Orthographic Projection .....	265
Tomasi-Kanade Factorization.....	268
Bundle Adjustment .....	270
SfM – Example Exercise.....	273
Object Tracking.....	274
Self-Supervised .....	284
Preliminaries .....	284
Definition .....	284
Importance .....	284
Contrastive Learning.....	296
Intuition .....	296
Terminology .....	297
Training Objectives .....	298
Design Choices for Contrastive Learning .....	300

Types of Learning.....	302
Coordinate-Based Networks.....	305
Implicit Neural Representations.....	305
Differentiable Volumetric Rendering.....	306
NeRF.....	307
Image Enhancement.....	309
Generative Models .....	309
Generative Adversarial Networks.....	310
Applications .....	314
Challenges.....	318
Super-Resolution Convolutional Neural Network (SRCNN).....	320
Autoencoders.....	323
U-NET .....	<b>Error! Bookmark not defined.</b>

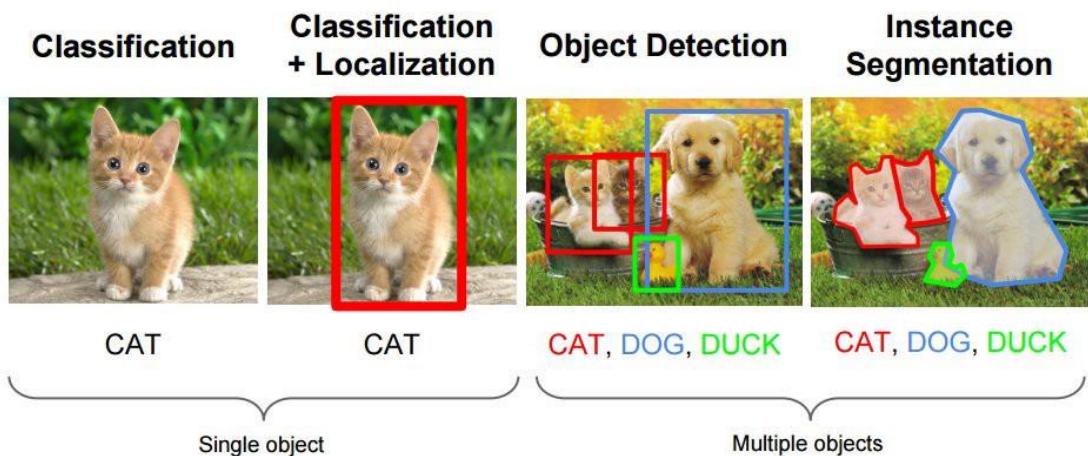
## COURSE AGENDA

Computer vision is the field of computer science that focuses on replicating parts of the complexity of the human vision system and enabling computers to identify and process objects in images and videos in the same way that humans do.

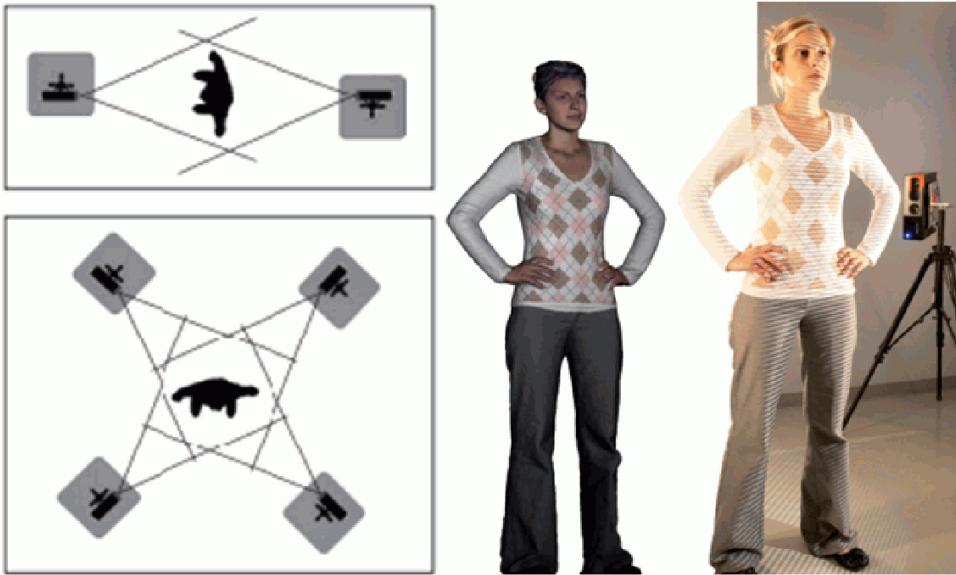
In our course, we will split the topic into 3 sections:

- Recognition - the classical problem in computer vision, image processing, and machine vision is that of determining whether the image data contains some specific object, feature, or activity. Different varieties of recognition problem are described in the literature.
  - Object Recognition (object classification) - one or several pre-specified or learned objects or object classes can be recognized, usually together with their 2D positions in the image or 3D poses in the scene.
  - Identification - an individual instance of an object is recognized. Examples include identification of a specific person's face or fingerprint, identification of handwritten digits, or identification of a specific vehicle.
  - Detection - the image data are scanned for a specific condition. Examples include the detection of possible abnormal cells or tissues in medical images or the detection of a vehicle in an automatic road toll system. Detection based on relatively simple and fast computations is sometimes used for finding smaller regions of interesting image data which can be further analyzed by more computationally demanding techniques to produce a correct interpretation.

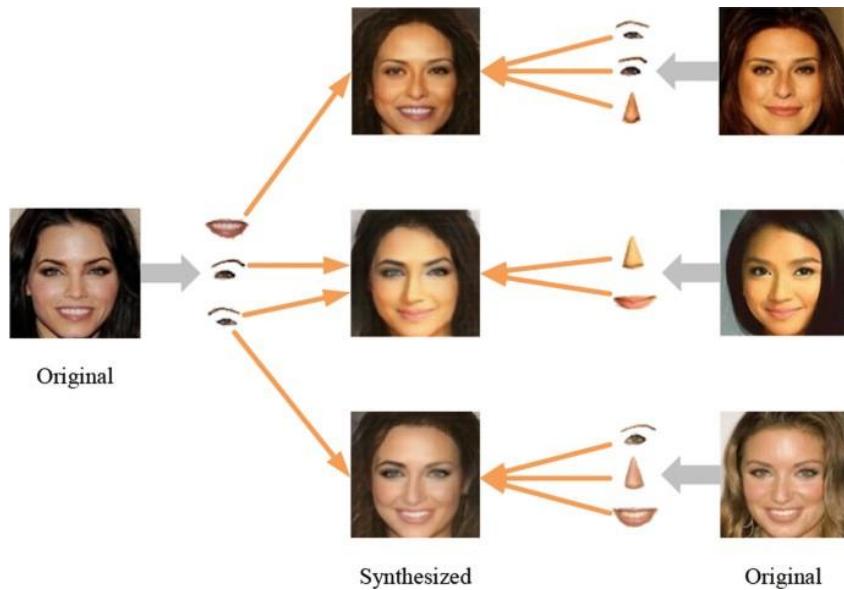
## Computer Vision Tasks



- Measurement - Dimension inspection is a type of appearance inspection. It plays an important role in making pass/fail judgments of whether parts or products have been worked or assembled according to specifications.



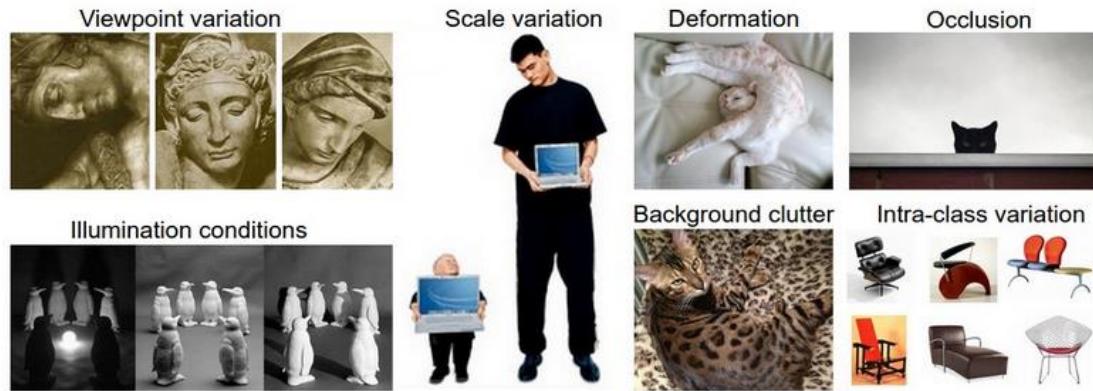
- Synthesis - aims to create new views of a specific subject starting from a number of pictures taken from given point of views.



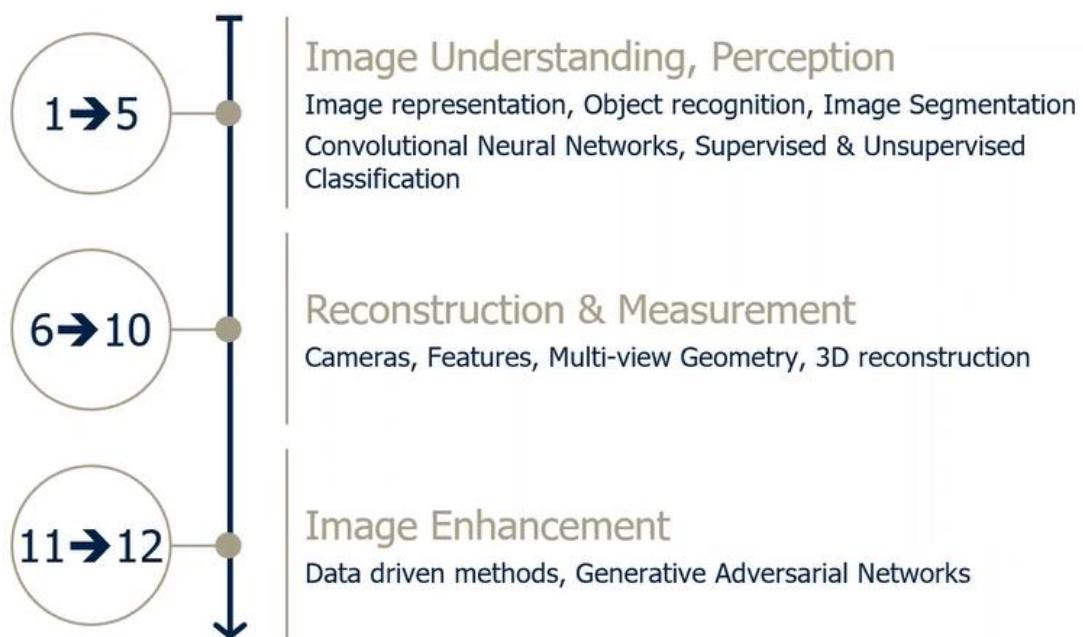
It is a very difficult problem, even for the most advanced neural network. The main challenges are:

- Viewpoint - The same instance of an object can be captured from different angles.
- Scale variation – Objects belonging to the same class often vary in scale.
- Deformation - Many objects can be deformed or change shape, even in extreme ways.
- Illumination conditions - The amount of illumination changes pixel values.

- Background clutter - Objects may blend into the surrounding environment, thus making them hard to detect.
- Intra-class variation - Objects within the same class can vary broadly in terms of appearance.



The reason why these are challenges is that they all change the matrix representation of the image and, especially, the instance of the object in the image that the computer sees. This is because the matrix representation of an image is nothing but a 2D or 3D (grayscale or RGB) matrix of brightness values.



## ETHICS IN COMPUTER VISION

Because computer vision enables artificial intelligence systems to identify faces, objects, locations, and movements, this technology raises a variety of ethical concerns and privacy issues. These include fraud, bias, inaccuracy, and the lack of informed consent.

For example:

- Fraud – Hackers and fraudsters have outsmarted facial recognition technology using masks and photos to claim benefits or gain entry to a site under another person's identity.
- Bias – In law enforcement and other arenas, facial recognition produces a far higher rate of false identifications among Black and Asian faces than white ones, increasing the chances of false arrest. It is also far more likely to identify elderly people and young children than middle-aged adults, skewing the evidence and affecting investigations.

### Twitter finds racial bias in image-cropping AI

© 20 May 2021



## Twitter's racist algorithm is also ageist, ableist and Islamaphobic, researchers find

Twitter hosted an effort at Def Con to identify more flaws in the automated system, which went viral after it was found to focus on white people and crop out Black people.



GETTY IMAGES  
Preferences for white people over black people and women over men were found in testing

Twitter's automatic cropping of images had underlying issues that favoured white individuals over black people, and women over men, the company said.

- Inaccuracy – In healthcare and disease detection, extraneous signals, or data noise, can lead to inaccuracy and faulty diagnoses.
- Legal Content Violations – In the private sector, facial recognition has been used to collect personal data without consent, resulting in violations of privacy laws, prompting a multitude of class action lawsuits.
- Ethical Content Violations – Researchers amass large data sets of facial images without consent, and this data, in addition to data scraped from the web, is often used to improve military and commercial surveillance algorithms. Unbeknownst to users, the personal data they post on the web is later used as training data for surveillance applications around the world.

## How to use computer vision ethically

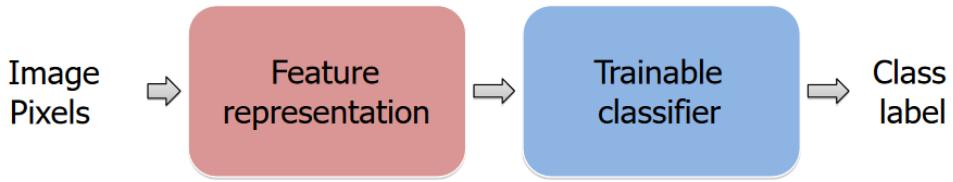
Because computer vision is a new technology and its implications are still poorly understood, there are few government regulations, institutional review procedures, and ethical best practices to guide and constrain its use. Until such guidelines exist, organizations, researchers, and private users must establish their own ethical frameworks for the use of computer vision. Below are some ways to get started.

Broad strategies for using computer vision ethically:

- Improve the training data - Biases and discrimination originate in the training data that is fed into a machine learning system. Careful curation and annotation of the data with a view toward diversity and objectivity, as well as stronger verification procedures and countermeasures, will make a model less prone to bias and discrimination.
- Choose the appropriate level of technology for the problem - When the capability of the technology exceeds the requirements of the application, unintended consequences follow and ethical risks increase. For example, a camera system designed to track the number of people entering and exiting a theme park would not need to use facial recognition technology.
- Clearly define the purpose for which the technology will be used - Set boundaries for the use of a specific CV model, document the execution of the model, and work to ensure that its application does not extend beyond its stated purpose and intent.
- Create and/or review strong internal privacy and data protection programs - Organizations need to ensure compliance with existing (and evolving) local laws as part of a larger effort to protect personal and client data from unintended distribution and use.
- Prioritize informed consent - To the fullest extent possible, obtain informed consent before collecting facial images and personal data. This is essential, and often mandated by law. In large scientific studies, consent could be obtained from a panel of representatives who can speak for a large population.

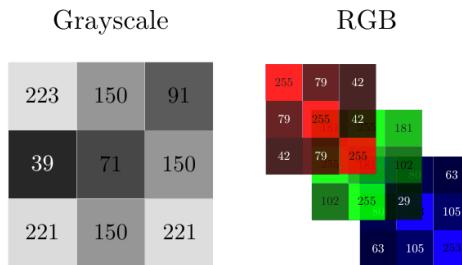
## IMAGE UNDERSTANDING

### Image Representation



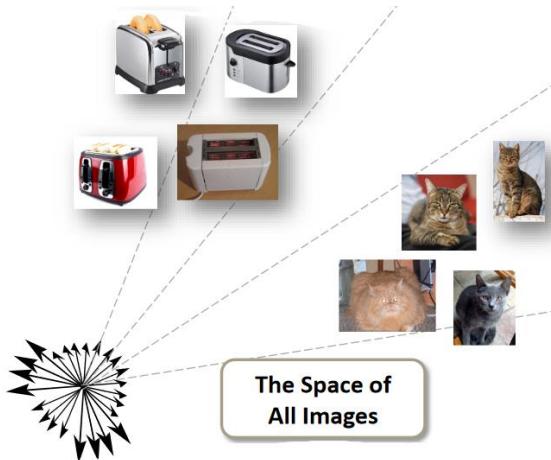
- Hand-crafted feature representation
- Off-the-shelf trainable classifier

An image is just a bunch of numbers, image can be a matrix or matrices of pixels. In the case of a grayscale image, this matrix will be made of numbers between 0 and 255. For RGB images, we'll have three matrices, one of each color channel. For instance:



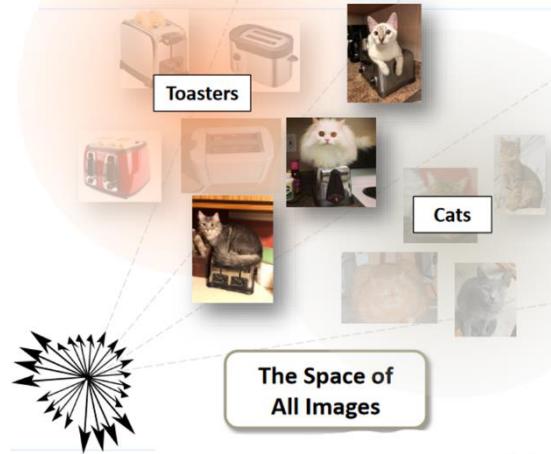
We stack all the numbers into a vector and then our training data is just a bunch of high-dimensional points now.

If we take an image that is  $10 \times 10$ , so are vectors are  $100 \times 1$ , and if we have 3 colors we have  $300 \times 1$  (or  $100 \times 1 \times 3$ ).



It's a bad representation, because we want for similar images to be close by. If we just change the background of the images or their color, then the images will be far away from each other.

Another method is to divide the space into different regions for different classes and define a distribution over space for each class.



For an image that we make with our phone, that is  $4032 \times 3024$  pixels size, and each pixel has 3 colors, we get 36,578,304 pixels (36.5 *Mega* pixels). But in practice, images sit on a lower dimensional manifold. So, we can think of image features and dimensionality reduction as ways to represent images by their location on such manifolds.

#### Pros:

- Easy to compute.

#### Cons:

- Sensitive to all types of transformations: color, lighting, translation, deformation, etc.

The simplest idea for an image classification model is to use the **nearest neighbor paradigm**. In particular, we first define a distance function:

$$d(I_1, I_2) = \sum_P |I_1(p) - I_2(p)|_1$$

Then, given a query image  $I$ , we find its nearest neighbor in our dataset:

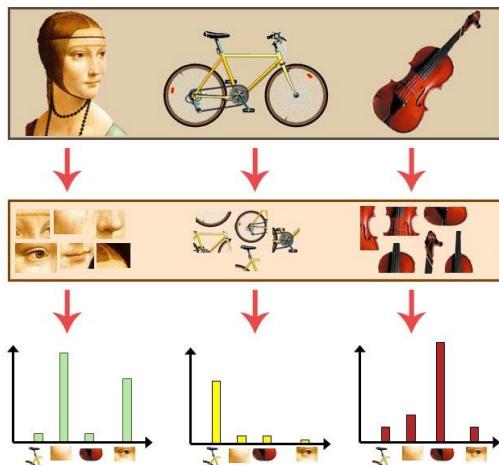
$$I^* = \arg \min_{I' \in D} d(I, I')$$

Finally, we classify the query image  $I$  with the class of  $I^*$ . Note that this algorithm is very slow, since we have to perform a nearest-neighbor search at test time (although, there is no training phase). Furthermore, this is a bad classifier, since **pixel distances are extremely uninformative**. In order to demonstrate the weakness of the classifier, consider two checkerboard patterns, where one is just a horizontal displacement of the other by one field. Even though, semantically, they are the same object, according to our method described above, the two images have maximal distance.

## "BAG OF FEATURES" REPRESENTATION

**Bag-of-features** (BoF) (also known as bag-of-visual-words) is a method to represent the features of images (i.e. a feature extraction/generation/representation algorithm). BoF is inspired by the bag-of-words model often used in the context of NLP, hence the name. In the context of computer vision, BoF can be used for different purposes, such as content-based image retrieval (CBIR), i.e. find an image in a database that is closest to a query image.

The general idea of **bag of visual words** (BOVW) is to represent an image as a set of **features**. Features consists of **keypoints** and **descriptors**. **Keypoints** are the “stand out” **points in an image**, so no matter the image is rotated, shrink, or expand, its keypoints will always be the same. And **descriptor** is **the description of the keypoint**. We use the keypoints and descriptors to construct **vocabularies** and represent each image as a **frequency histogram of features** that are in the image. From the frequency histogram, later, we can find another similar images or predict the category of the image.



### Steps

The BoF can be divided into three different steps. To understand all the steps, consider a training dataset  $D = \{x_1, x_2, \dots, x_N\}$  of  $N$  training images. Then BoF proceeds as follows.

#### 1. Feature extraction

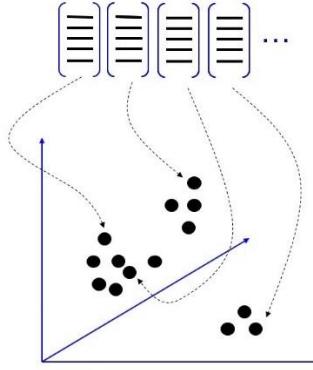
In this first step, we extract all the *raw* features (i.e. keypoints and descriptors) from all images in the training dataset  $D$ . This can be done with **SIFT** (Scale-Invariant Feature Transform), where each descriptor is a 128-dimensional vector that represents the neighborhood of the pixels around a certain keypoint (e.g. a pixel that represents a corner of an object in the image).

Note that image  $x_i \in D$  may contain a different number of **features (keypoints and descriptors)** than image  $x_j \neq x_i \in D$ . As we will see in the third step, BoF produces a feature vector of size  $k$  for all images, so all images will be represented by a fixed-size vector.

Let  $F = \{f_1, f_2, \dots, f_M\}$  be the set of descriptors extracted from all training images in  $D$ , where  $M \gg N$ . So,  $f_i$  may be a descriptor that belongs to any of the training examples (it does not matter which training image it belongs to).

## 2. Learn “visual vocabulary”

In this step, we cluster all descriptors  $F = \{f_1, f_2, \dots, f_M\}$  into  $k$  clusters using **k-means** (or another clustering algorithm like DBSCAN). This is sometimes known as the **vector quantization (VQ)** step. In fact, the idea behind VQ is very similar to clustering and sometimes VQ is used interchangeably with clustering.



## 3. Quantize local features using visual vocabulary

So, after this step, we will have  $k$  clusters, each of them associated with a centroid  $C = \{c_1, \dots, c_k\}$ , where  $C$  is the set of centroids (and  $c_i \in \mathbb{R}^{128}$  in the case that SIFT descriptors have been used). These centroids represent the main features that are present in the whole training dataset  $D$ . In this context, they are often known as the **codewords** (which derives from the vector quantization literature) or **visual words** (hence the name bag-of-visual-words). The set of codewords  $C$  is often called **codebook** or, equivalently, the **visual vocabulary**.

## 4. Represent images by frequency of “visual words”

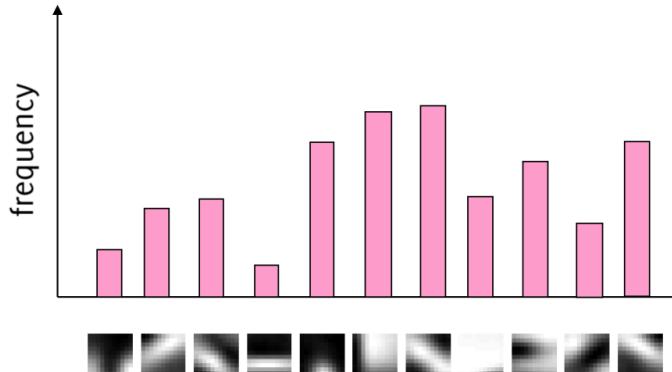
In this last step, given a new (test) image  $u \notin D$  (often called the **query image** in this context of CBIR), then we will represent  $u$  as a  **$k$ -dimensional vector** (where  $k$ , if you remember, is the number of codewords) that will represent its **feature vector**. To do that, we need to follow the following steps.

1. Extract the raw features from  $u$  with e.g. SIFT (as we did for the training images). Let the descriptors of  $u$  be  $U = \{u_1, u_2, \dots, u_{|U|}\}$ .
2. Create a vector  $I \in \mathbb{R}^k$  of size  $k$  filled with zeros, where the  $i$ th element of  $I$  corresponds to the  $i$ th codeword (or cluster).
3. For each  $u_i \in U$ , find the *closest* codeword (or centroid) in  $C$ . Once you found it, increment the value at the  $j$ th position of  $I$  (i.e., initially, from zero to one), where  $j$  is the found closest codeword to the descriptor  $u_i$  of the query image.

The distance between  $u_i$  and any of the codewords can be computed e.g. with the Euclidean distance. Note that the descriptors of  $u$  and the codewords have the same dimension because they have been computed with the same feature descriptor (e.g. SIFT).

At the end of this process, we will have a vector  $I \in \mathbb{R}^k$  that represents the **frequency of the codewords in the query image  $u$**  (akin to the *term frequency* in the context of the bag-of-

words model), i.e.  $u$ 's feature vector. Equivalently,  $I$  can also be viewed as a histogram of features of the query image  $u$ . Here's an illustrative example of such a histogram.



From this diagram, we can see that there are 11 codewords (of course, this is an unrealistic scenario!). On the  $y$ -axis, we have the frequency of each of the codewords in a given image. We can see that the 7th codeword is the most frequent in this particular query image.

Alternatively, rather than the codeword frequency, we can use the [tf-idf \(explanation\)](#). In that case, each image will be represented not by a vector that contains the frequency of the codewords but it will contain the frequency of the codewords weighted by their presence in other images.

### Conclusion

To conclude, BoF is a method to represent features of an image, which could then be used to train classifiers or generative models to solve different computer vision tasks (such as [CBIR](#)). More precisely, if you want to perform CBIR, you could compare your query's feature vector with the feature vector of every image in the database, e.g. using the cosine similarity.

The first two steps above are concerned with the creation of a **visual vocabulary** (or codebook), which is then used to create the feature vector of a new test (or query) image.

#### Pros:

- Robust to geometrical transformations.
- Robust to color transformations.

#### Cons:

- Does not capture geometry.
- Depends on quality of the dictionary of the visual words.

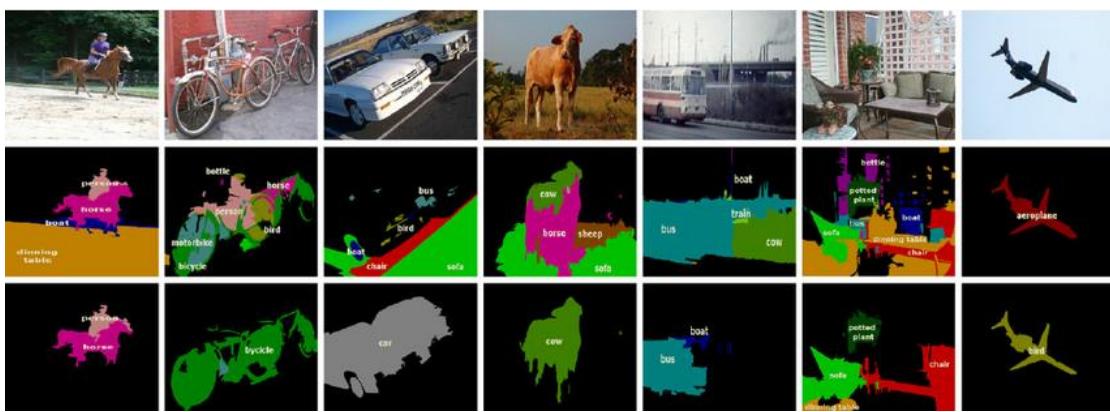
#### A side note

As a side note, the term bag is used because the (relative) order of the features in the image is lost during this feature extraction process, and this can be a disadvantage.

## DATA SETS

### PASCAL VOC

The PASCAL Visual Object Classes (VOC) 2012 dataset contains **20 object categories** including vehicles, household, animals, and other: aero plane, bicycle, boat, bus, car, motorbike, train, bottle, chair, dining table, potted plant, sofa, TV/monitor, bird, cat, cow, dog, horse, sheep, and person. Each image in this dataset has pixel-level segmentation annotations, bounding box annotations, and object class annotations. This dataset has been widely used as a benchmark for object detection, semantic segmentation, and classification tasks. The PASCAL VOC dataset is split into three subsets: 1,464 images for training, 1,449 images for validation and a private testing set.

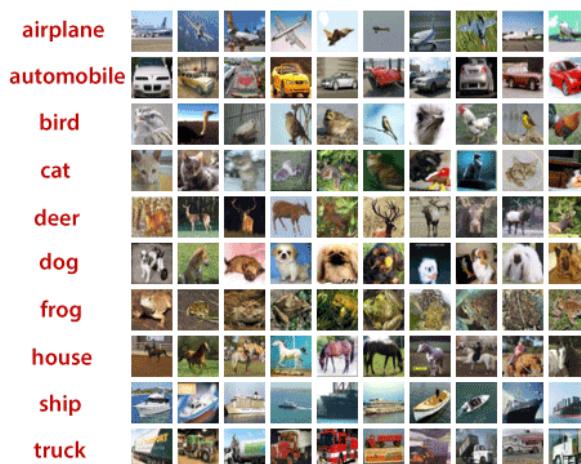


### CIFAR-10 & CIFAR-100

The CIFAR-10 and CIFAR-100 are labeled subsets of the 80 million tiny images.

CIFAR-10 contains 60000 32x32 color images with 10 classes (animals and real-life objects). There are 6000 images per class. This dataset has 50000 training images and 10000 test images. The classes are mutually exclusive, without any overlaps.

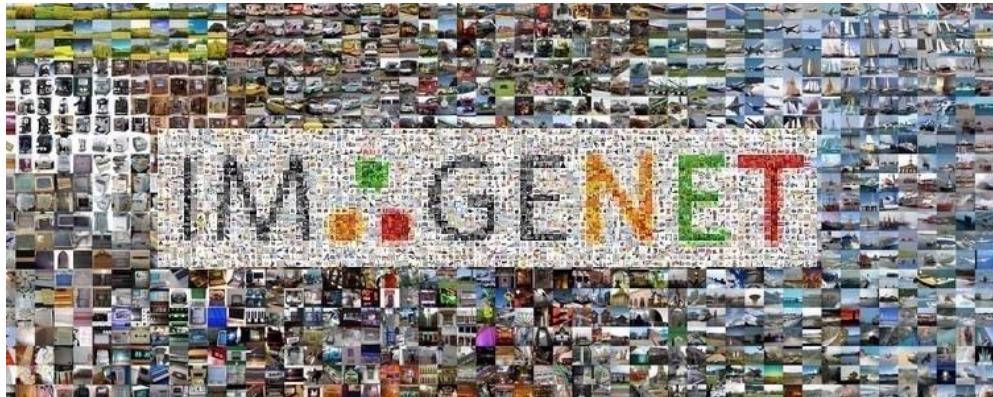
CIFAR-100 consists of 100 classes containing 600 images each. There are 500 training images and 100 testing images per class.



## ImageNet

ImageNet is one of the most popular image databases with more than **14 million hand-annotated images**.

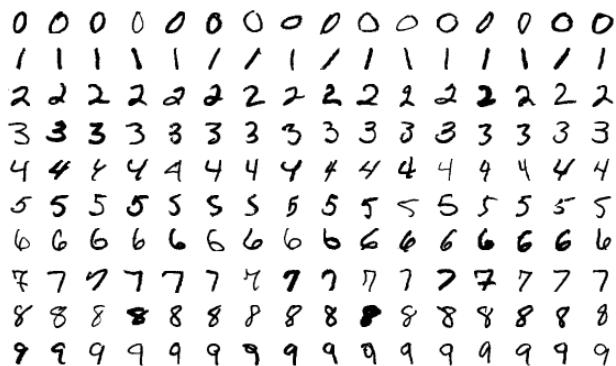
This database is organized according to the WordNet hierarchy (currently only the nouns), in which hundreds and thousands of images depict each node of the hierarchy. Object-level annotations provide a bounding box around the (visible part of the) indicated object.



## MNIST

It's a large database of **handwritten single digits** containing 60,000 training images and 10,000 testing images.

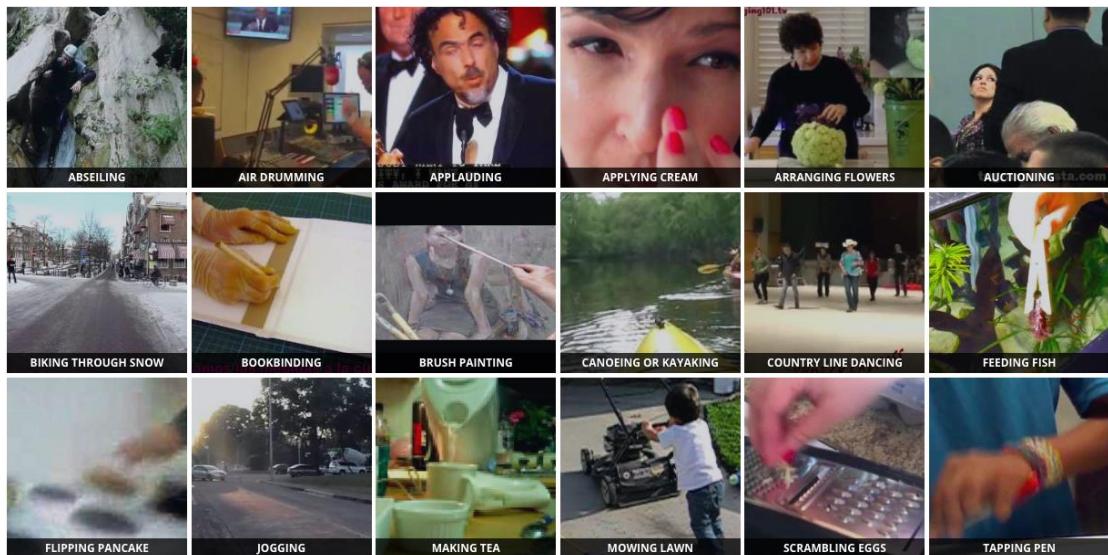
It was released in 1998 and is used for classification tasks.



## Kinetics-700

It is a large **video dataset** consisting of 650,000 clips covering 700 human action classes.

The videos include human-object interactions like playing instruments and human-human interactions like hugging. Each action class has at least 700 video clips, and each clip is annotated with an action class lasting for about 10 seconds.

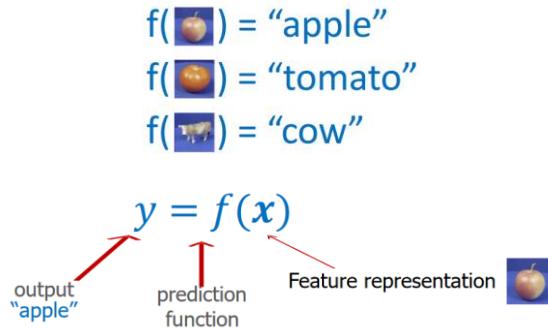


## CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks are a type of an Artificial Neural Network model particularly suitable for image recognition. As we discussed in the previous section, these models can be learnt end-to-end, thus solving the problem of having to hand-engineer features and components.

### Statistical Learning Approach

Apply a prediction function to a feature representation of the image to get the desired output:



The input can be an image or some representation of the image.

The general way to look at this problem is to split it into training and testing.

#### Training

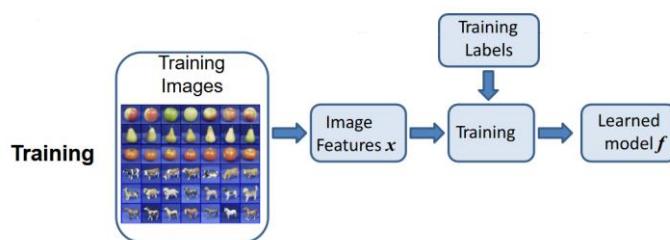
Given labeled *training set*  
 $\{(x_1, y_1), \dots, (x_n, y_n)\}$

Learn the prediction function  
 $f$ , by minimizing prediction error on *training set*

#### Testing

Given unlabeled *test instance*  $x$   
Predict the output label  $y$  as  $y = f(x)$

### STEPS



In training, we have a set of "Training Images". We can extract some features from the images, or those could be raw RGB pixels. The features go into the training stage, where we also have the training labels the  $y$ 's, and we learn the model  $f$ .

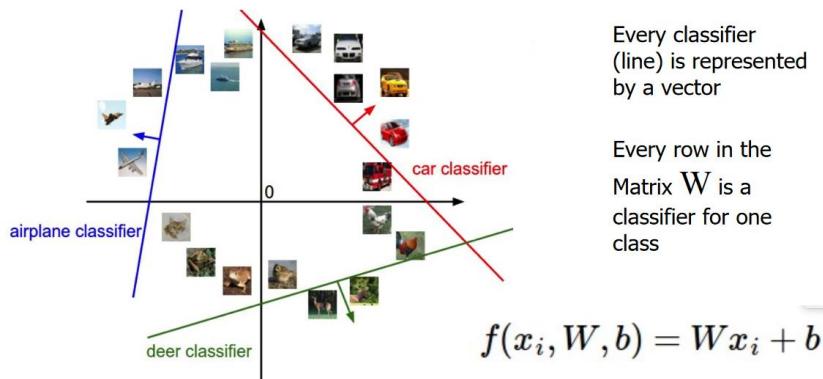


In testing, we just give the test image. We extract the same features, it goes into the function  $f$ , that we have learned, and we get the output.

**Linear classifiers** classify data into labels based on a linear combination of input features.

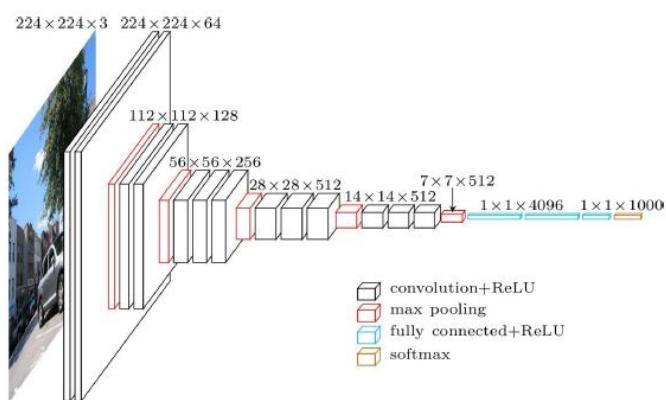
$$f(x_i, W, b) = Wx_i + b$$

Therefore, these classifiers separate data using a line or plane or a hyperplane. They can only be used to classify data that is linearly separable. They can be modified to classify non-linearly separable data.



The various deep learning methods use data to train neural network algorithms to do a variety of machine learning tasks, such as the classification of different classes of objects. **Convolutional neural networks (CNN)** are deep learning algorithms that are very powerful for the analysis of images.

These models typically have three types of layers: convolutional layers, downsampling layers, and fully connected layers. The model takes an input image (for ex.  $224 \times 224 \times 3$ ), and **successively decrease the spatial dimensions** as the image is passed through the network.



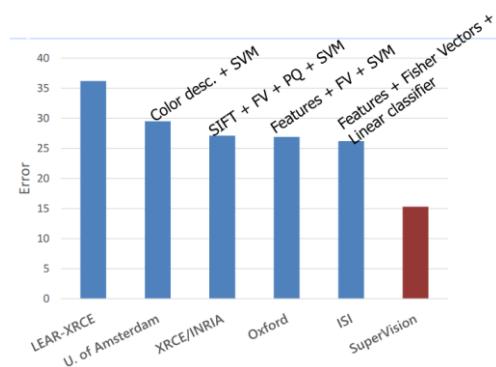
At the same time, the model increases the number of feature channels in order to **improve the expressiveness**.

At the end, typically there is a  $1 \times K$  dimensional vector (where  $K$  is the number of classes to be recognized), upon which we utilize the softmax operator in order to obtain a valid probability distribution over the classes.

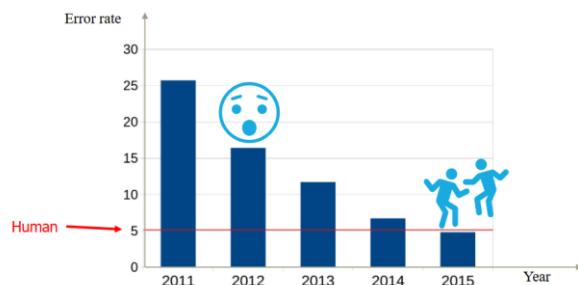
## History

- ▶ **1950's:** neural nets (perceptron) invented by Rosenblatt
- ▶ **1980's/1990's:** Neural nets are popularized and then abandoned as being interesting idea but impossible to optimize or "unprincipled"
- ▶ **1990's:** LeCun achieves state-of-art performance on character recognition with convolutional network (main ideas of today's networks)
- ▶ **2000's:** Hinton, Bottou, Bengio, LeCun, Ng, and others keep trying stuff with deep networks but without much traction/acclaim in vision
- ▶ **2010-2011:** Substantial progress in some areas, but vision community still unconvinced
  - Some neural net researchers get ANGRY at being ignored/rejected
- ▶ **2012:** shock at ECCV 2012 with ImageNet challenge

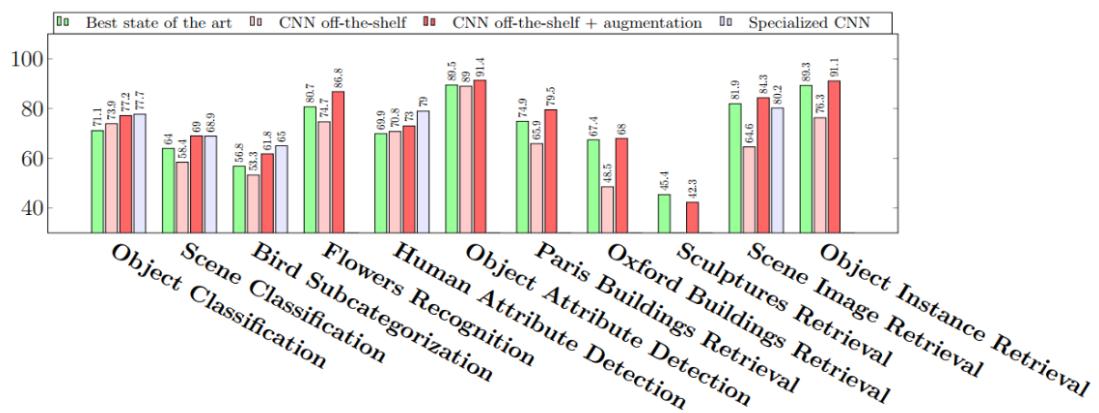
At 2012 4 out of 5 models that was purposed to the ImageNet challenge was based on linear classifiers, and had a classification error of more than 25%, except of the model that was based on CNN, that had an error of 15%.



This model was upgraded and in 2015 it had a mean error rate below the human error.



A group from university of Stockholm, took one of the SOTA CNN, that was used for image classification, and applied it on different tasks.



## Intro to CNNs

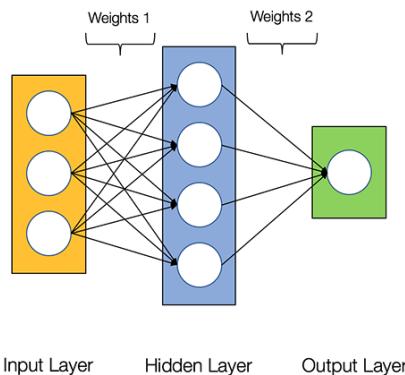
### NEURAL NETWORK

Neural Networks is basically a mathematical function that maps a given input to a desired output.

Neural Networks consist of the following components:

- An **input layer** –  $x$
- An arbitrary amount of **hidden layers**
- An **output layer** -  $\hat{y}$  (**predictor** or the **estimator**)
- A set of **weights** and **biases** between each layer,  $W$  and  $b$
- A choice of **activation function** for each hidden layer,  $\sigma$ . Here, we'll use a **Sigmoid** activation function.

The diagram below shows the architecture of a 2-layer Neural Network (*note that the input layer is typically excluded when counting the number of layers in a Neural Network*).



### *Training the Neural Network*

The output  $\hat{y}$  of a simple 2-layer Neural Network is:

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

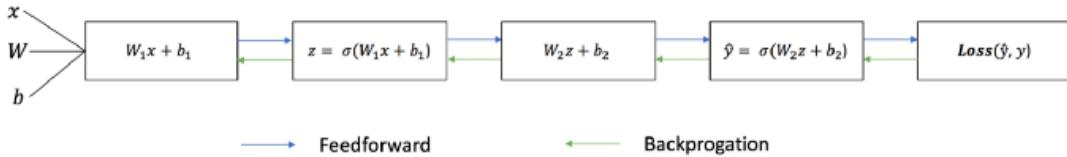
You might notice that in the equation above, the weights  $W$  and the biases  $b$  are the only variables that affects the output  $\hat{y}$ .

Naturally, the right values for the weights and biases determines the strength of the predictions. The process of fine-tuning the weights and biases from the input data is known as **training the Neural Network**.

Each iteration of the training process consists of the following steps:

- Calculating the predicted output  $\hat{y}$ , known as **feedforward**
- Updating the weights and biases, known as **backpropagation**

The sequential graph below illustrates the process:



### Feedforward

As we've seen in the sequential graph above, **feedforward** is just simple calculus and for a basic 2-layer neural network, the output of the Neural Network is:

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

However, we still need a way to evaluate the “goodness” of our predictions (i.e. how far off are our predictions)? The **Loss Function** allows us to do exactly that.

### Loss Functions

There are many available loss functions, and the nature of our problem should dictate our choice of loss function. Here, we'll use a simple **sum-of-squares error** as our loss function.

$$\text{Sum of squares error} = \sum_{i=1}^n (y - \hat{y})^2$$

That is, the sum-of-squares error is simply the sum of the difference between each predicted value and the actual value. The difference is squared so that we measure the absolute value of the difference.

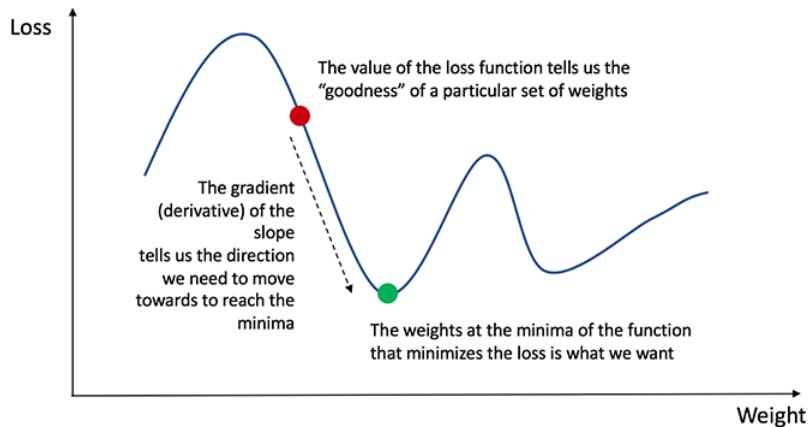
**Our goal in training is to find the best set of weights and biases that minimizes the loss function.**

### Backpropagation

Now that we've measured the error of our prediction (loss), we need to find a way to **propagate** the error back, and to update our weights and biases.

In order to know the appropriate amount to adjust the weights and biases by, we need to know the **derivative of the loss function with respect to the weights and biases**.

Recall from calculus that the derivative of a function is simply the slope of the function.



If we have the derivative, we can simply update the weights and biases by increasing/reducing with it (refer to the diagram above). This is known as **gradient descent**.

However, we can't directly calculate the derivative of the loss function with respect to the weights and biases because the equation of the loss function does not contain the weights and biases. Therefore, we need the **chain rule** to help us calculate it.

$$\begin{aligned}
 Loss(y, \hat{y}) &= \sum_{i=1}^n (y - \hat{y})^2 \\
 \frac{\partial Loss(y, \hat{y})}{\partial W} &= \frac{\partial Loss(y, \hat{y})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial W} = ; \text{where } z = Wx + b \\
 &= 2(y - \hat{y}) \cdot (\text{derivative of the activation function}) \cdot x = ; \text{activ. is Sigmoid} \\
 &= 2(y - \hat{y}) \cdot z(1 - z) \cdot x
 \end{aligned}$$

(For a deeper understanding of the application of calculus and the chain rule in backpropagation, I strongly recommend [this tutorial by 3Blue1Brown](#).)

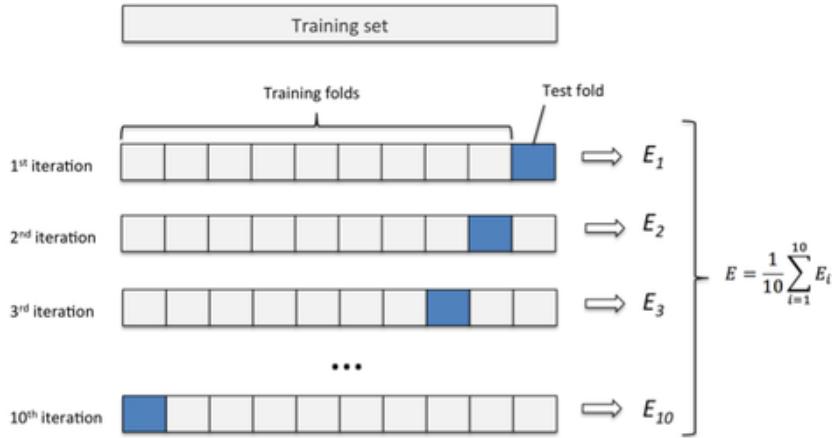
### Evaluation

After training the model the most important part is to evaluate the classifier to verify its applicability.

#### Holdout method

There are several methods exists and the most common method is the holdout method. In this method, the given data set is divided into 2 partitions as test and train 20% and 80% respectively. The train set will be used to train the model and the unseen test data will be used to test its predictive power.

## Cross-validation



Over-fitting is a common problem in machine learning which can occur in most models.  $k$ -fold cross-validation can be conducted to verify that the model is not over-fitted. In this method, the data-set is randomly partitioned into  $k$  *mutually exclusive* subsets, each approximately equal size and one is kept for testing while others are used for training. This process is iterated throughout the whole  $k$  folds.

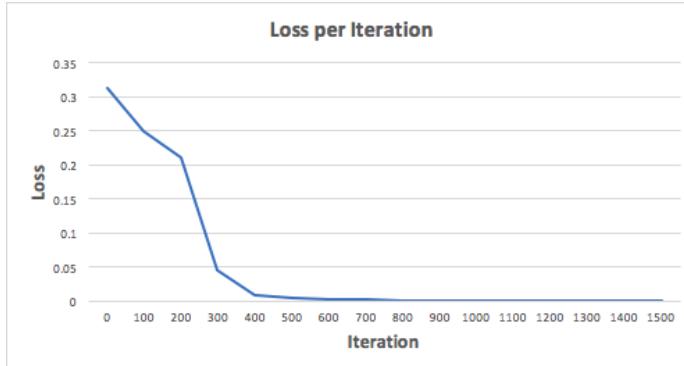
*Putting it all together*

Now let's apply our Neural Network on an example and see how well it does.

X1	X2	X3	Y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Our Neural Network should learn the ideal set of weights to represent this function. Note that it isn't exactly trivial for us to work out the weights just by inspection alone.

Let's train the Neural Network for 1500 iterations and see what happens. Looking at the loss per iteration graph below, we can clearly see the loss **monotonically decreasing towards a minimum**. This is consistent with the gradient descent algorithm that we've discussed earlier.



Let's look at the final prediction (output) from the Neural Network after 1500 iterations.

Prediction	Y (Actual)
0.023	0
0.979	1
0.975	1
0.025	0

We did it! Our feedforward and backpropagation algorithm trained the Neural Network successfully and the predictions converged on the true values.

Note that there's a slight difference between the predictions and the actual values. This is desirable, as it prevents **overfitting** and allows the Neural Network to **generalize** better to unseen data.

There are different types of activation functions, like:

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) \stackrel{x \neq 0}{=} \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

## CNN

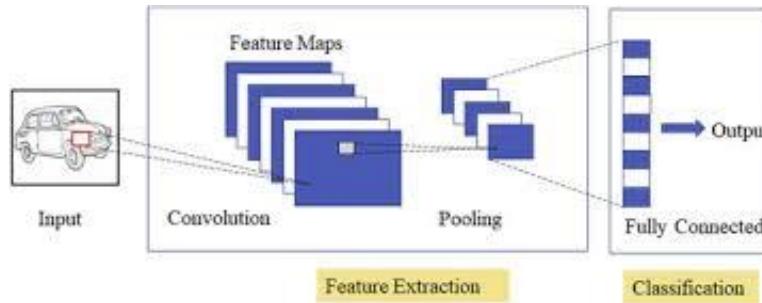
CNN is mainly used in image analysis tasks like image recognition, object detection and segmentation.

### INPUT LAYER

*Number of parameters learned*

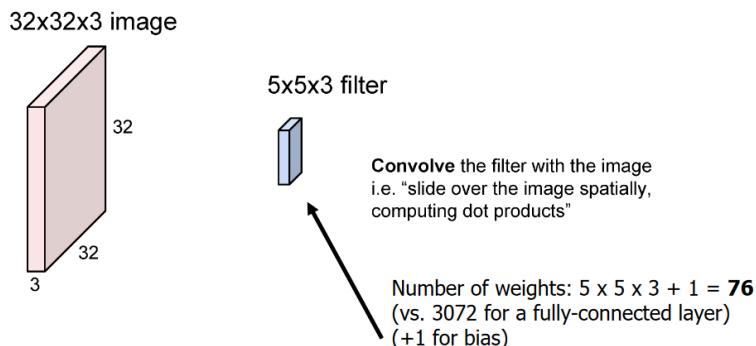
Input layer has nothing to learn, at its core, what it does is just provide the input image's shape. So no learnable parameters here. Thus **number of parameters = 0**.

After the input layer, there are three types of layers in CNNs:



### CONVOLUTIONAL LAYER

The **convolutional layer** is the **core building block of every CNN**. In each layer, we have a set of learnable filters. **We convolve the input with each filter during forward propagation, producing an output activation map of that filter**. As a result, the network learns activated filters when specific features appear in the input image.



In fully connected layers each neuron in the successive layer is connected to all neurons in the previous layer. On the other hand, in convolutional layers each neuron in the successive layer is connected only to a small neighborhood in the previous layer. Furthermore, each neuron in a given layer (and a given channel) **shares the weights**, thus **drastically reducing the number of parameters**. Convolutional layers are **translation equivariant**:

$$\mathcal{T}_\theta[f](H) = f(\mathcal{T}_\theta[H])$$

Meaning that the order of applying a translation  $\mathcal{T}_\theta$  and convolution  $f$  doesn't matter. In other words, if we translate the input to the convolutional layer, then the output of the convolution gets translated in the same way.

### Hyperparameters

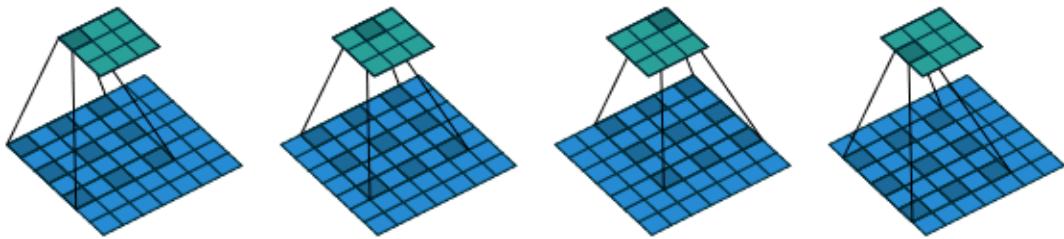
To formulate a way to compute the output size of a convolutional layer, we should first discuss two critical hyperparameters.

#### Dilation rate (Atrous)

The **dilation rate** defines a **spacing between the values in a kernel**. A  $3 \times 3$  kernel with a dilation rate of 2 will have the same field of view as a  $5 \times 5$  kernel, while only using 9 parameters. Imagine taking a  $5 \times 5$  kernel and deleting every second column and row.

The **dilation “rate”** is controlled by an additional hyperparameter  $d$ . Implementations may vary, but there are usually  $d - 1$  spaces inserted between kernel elements such that  $d = 1$  corresponds to a regular convolution.

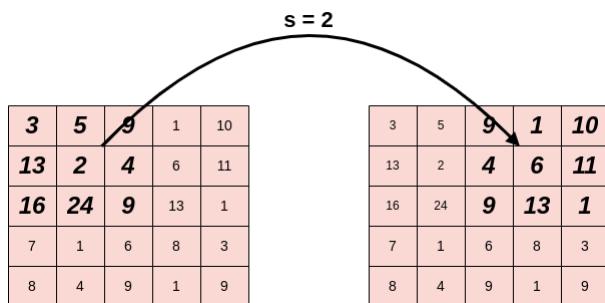
This delivers a wider field of view at the same computational cost. Dilated convolutions are particularly popular in the field of real-time segmentation. Use them if you need a wide field of view and cannot afford multiple convolutions or larger kernels.



#### Stride

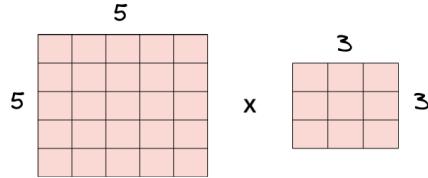
During convolution, the filter slides from left to right and from top to bottom until it passes through the entire input image. We define stride  $S$  as the step of the filter. So, when we want to downsample the input image and end up with a smaller output, we set  $S > 1$ .

Below, we can see an example when  $S = 2$ :



#### Padding

In a convolutional layer, we observe that **the pixels located on the corners and the edges are used much less than those in the middle**. For example, in the below example, we have a  $5 \times 5$  input image and a  $3 \times 3$  filter:



Below we can see the times that each pixel from the input image is used when applying convolution with  $S = 1$ :

1	2	3	2	1
2	4	6	4	2
3	6	9	6	3
2	4	6	4	2
1	2	3	2	1

We can see that the pixel (0,0) is used only once while the central pixel (3,3) is used nine times.

In general, pixels located in the middle are used more often than pixels on edges and corners.

**Therefore, the information on the borders of images is not preserved as well as the information in the middle.**

Another problem is that the output of the layers is shrunk in comparison to the input. For a grayscale image of size  $N \times N$  with  $f \times f$  filter size the output size is  $(N - f + 1) \times (N - f + 1)$

A simple and powerful solution to this problem is **padding**, which adds rows and columns of zeros to the input image. If we apply padding  $P$  in an input image of size  $W \times H$ , the output image has dimensions  $(W + 2P) \times (H + 2P)$ .

Below we can see an example image before and after padding with  $P = 2$ . As we can see, the dimensions increased from  $5 \times 5$  to  $9 \times 9$ :

3	5	9	1	10		0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	2	4	6	11		0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	24	9	13	1	padding	0	0	3	5	9	1	10	0	0	0	0	0	0	0
7	1	6	8	3		0	0	13	2	4	6	11	0	0	0	0	0	0	0
8	4	9	1	9		0	0	16	24	9	13	1	0	0	0	0	0	0	0

By using padding in a convolutional layer, **we increase the contribution of pixels at the corners and the edges to the learning procedure.**

The usual padding dimension is:

$$P = \frac{K - 1}{2}$$

Where  $K$  is the kernel dimension.

### *Types of Padding*

#### Same Padding

In this type of padding, the **padding layers append zero values in the outer frame of the images** or data so the filter we are using can cover the edge of the matrix and make the inference with them too.

#### Valid Padding

This type of padding can be considered as **no padding**. We don't apply padding but we assume that every pixel of the image is valid so that the input can get fully covered by the filter wherein a simple model assumes corners are invalid and do not consider them in the coverage area.

### *Number of parameters learned*

This is where CNN learns, so certainly we'll have weight matrices. To calculate the learnable parameters here, all we have to do is just multiply the by the shape of **width  $M$ , height  $N$ , previous layer's filters  $d$**  and account for all such filters  **$k$  in the current layer**. Don't forget the bias term for each of the filter. Number of parameters in a CONV layer would be :

$$((M \cdot N \cdot d) + 1) \cdot k$$

We have added 1 because of the bias term for each filter. The same expression can be written as follows:

$$(Shape\ of\ width\ of\ the\ filter \cdot Shape\ of\ height\ of\ the\ filter \cdot Number\ of\ filters\ in\ the\ previous\ layer + 1) \cdot Number\ of\ filters$$

Where the term "filter" refer to the number of filters in the current layer.

### *Output Size*

We have the following input:

- An image of dimensions  $W_{in} \times H_{in}$
- A filter of dimensions  $K \times K$
- Stride  $S$  and padding  $P$

The output activation map will have the following dimensions:

$$W_{out} = \frac{W_{in} - K + 2P}{S} + 1 = \left\lfloor \frac{W_{in} - K + 2P}{S} + 1 \right\rfloor$$

$$H_{out} = \frac{H_{in} - K + 2P}{S} + 1 = \left\lfloor \frac{H_{in} - K + 2P}{S} + 1 \right\rfloor$$

If the output dimensions are not integers, it means that we haven't set the stride  $S$  correctly.

We have two exceptional cases:

- When there is no padding at all, the output dimensions:

$$\left( \frac{W_{in} - K}{S} + 1, \frac{H_{in} - K}{S} + 1 \right)$$

- In case we want to keep the size of the input unchanged after the convolution layer, we apply same padding where  $W_{out} = W_{in}$  and  $H_{out} = H_{in}$ . If  $S = 1$ , we set  $P = \frac{K-1}{2}$ .

Example

Let's suppose that we have an input image of size  $127 \times 127 \times 3$ , and 16 filters of size  $5 \times 5$ , padding  $P = 2$  and stride  $S = 2$ . Then the output dimensions are the following:

$$W_{out} = \frac{127 - 5 + 2 \cdot 2}{2} + 1 = 64$$

$$H_{out} = \frac{127 - 5 + 2 \cdot 2}{2} + 1 = 64$$

So, the output activation map will have dimensions:

$$64 \times 64$$

If we have 16 convolutions, hence we get the output size of:

$$64 \times 64 \times 16$$

And the number of parameters we have learned:

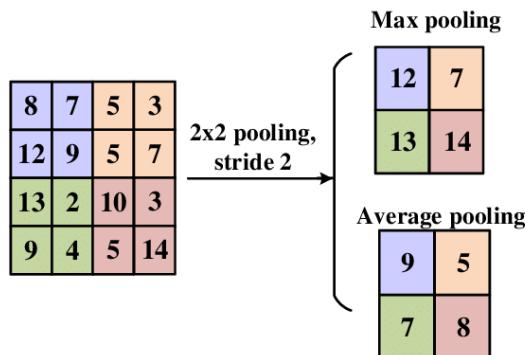
$$(5 \cdot 5 \cdot 3) + 1 \cdot 16$$

Sometimes we aren't interested in the bias, hence we get the number of parameters we learned:

$$5 \times 5 \times 3 \times 16 = 1200$$

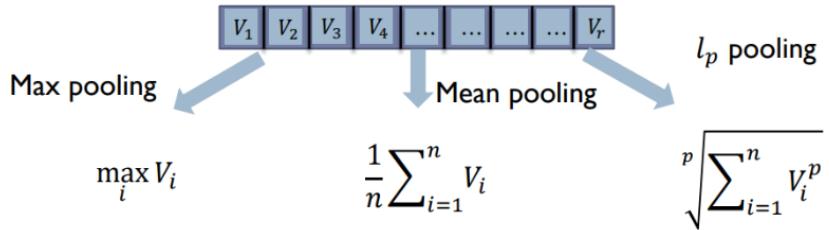
## POOLING LAYER

The **pooling layer** is used to **reduce the dimensionality of the feature map (downsampling)** and it makes feature detection more **robust to object orientation and scale changes** and increases **the receptive field**. There will be multiple activation and pooling layers inside the hidden layer of the CNN.



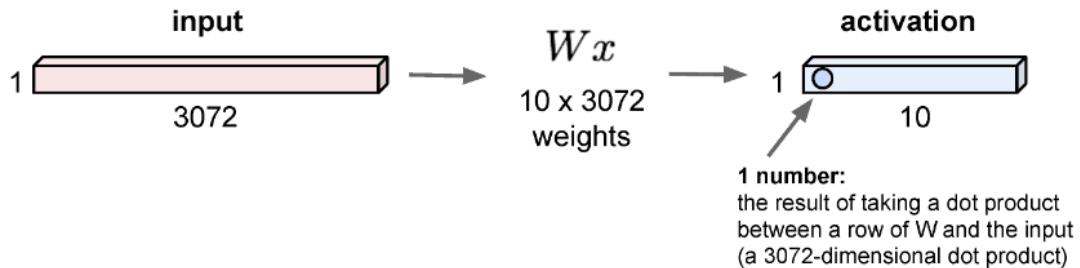
*Number of parameters learned*

This has got no learnable parameters because all it does is calculate a specific number, no backprop learning involved! Thus number of **parameters = 0**.

*Other Pooling Operators***FULLY-CONNECTED (FC) LAYER**

**Fully connected layers** form the last few layers in the network. The input to the fully connected layer is the output from the final pooling or convolutional layer, which is flattened and then fed into the fully connected layer.

32x32x3 image  $\rightarrow$  stretch to 3072 x 1

*Number of parameters learned*

This certainly has learnable parameters, matter of fact, in comparison to the other layers, this category of layers has the highest number of parameters, because, every neuron is connected to every other neuron. The number of parameters here is the product of the number of neurons in the current layer  $c$  and the number of neurons on the previous layer  $p$  and as always, do not forget the bias term. Thus number of parameters here are:

$$((c \cdot p + 1) \cdot c)$$

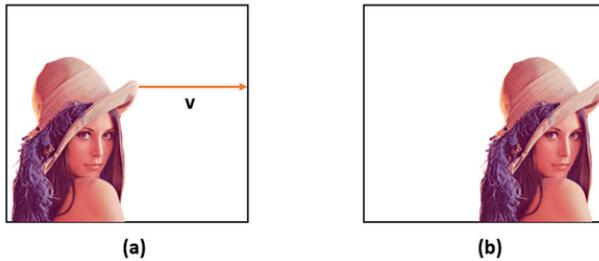
**TRANSLATION INVARIANCE AND EQUIVARIANCE***What is a Translation?*

A **translation** is a geometric transformation that shifts all points in a given direction and by the same distance. Alternatively, it can be interpreted as sliding the origin of the coordinate system by the same amount but in the opposite direction.

The translation can be expressed mathematically as the vector sum of a constant vector  $v$  to each point  $x$ :

$$T_v(x) = x + v$$

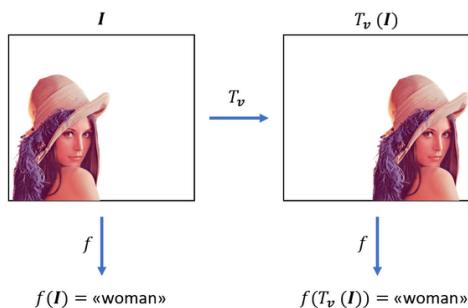
In the following example, image (b) was obtained from image (a) by shifting each pixel 150 pixels to the right:



### *Translation Invariance*

A certain property is **translation invariant** if it **doesn't change under any translation**. Let's consider the images above. We can recognize a woman in both images even though the pixel values were shifted. An image classifier should predict the label "woman" for both images. In fact, the classifier output should not be influenced by the position of the target. Hence, the output of the classifier function is translation invariant.

In the following figure, we illustrated schematically the translation invariance of an image classifier, whose classifier function is denoted as  $f$ :



### *Translation Equivariance*

Translation equivariance is a property often confused with translation invariance. So let's recall the mathematical **definition of equivariance**:

A function  $f$  is said equivariant to a function  $g$  if and only if:

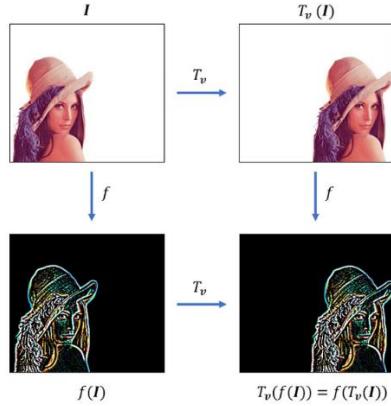
$$f(g(x)) = g(f(x))$$

In other words,  $f$  is equivariant to  $g$  if the order of application **does not change the result of the composite function**.

For example, the **convolution operation is translation equivariant**. In fact, if we apply a translation  $T_v$  to the image  $I$  and then perform a convolution  $f$ , the resulting response image

$f(T_v(i))$  will be the same as if we first performed convolution to  $I$  and then applied the same translation  $T_v$  to the response  $f(I)$ .

The translation equivariance of the convolution operation is represented below:



*Translation Invariance and Equivariance in Convolutional Neural Networks*  
 Translation invariance and equivariance are different properties of Convolutional Neural Networks (CNNs).

**The translation equivariance is obtained by means of the convolutional layers.** In fact, if the input image is translated to the right by a certain amount, the feature maps generated by convolutional layers are shifted by the same amount and direction.

CNNs designed for image classification are translation invariant since if we translate the input, then the output label will not be influenced. **Translation invariance is obtained in CNNs by means of the pooling layers.** The pooling operation is usually applied to the feature map generated by preceding convolutional layers and non-linear activation functions. Pooling is the substitution of features in a neighborhood with representative statistics, the max or the mean generally. Hence, the location of the original feature is disregarded.

**CNNs are not naturally equivariant and invariant to rotation, scaling, and affine transformations.** Hence, data augmentation techniques must be used to make CNNs more robust to such geometric transformations.

## Training

Let us now discuss a suitable choice for the output activations and the loss function, in order to successively train models for image classification.

In order to be able to predict classes (or categories), we need to make use of the **categorical distribution**. This is a discrete distribution, which for each class  $c$  assigns a probability value  $\mu_c$ :

$$p(y = c) = \mu_c$$

A more useful alternative notation is to represent the target vector  $y$  as a **one hot encoding**, such that  $y_c \in \{0,1\}$  (ex.  $y = (0, \dots, 0, 1, 0, \dots, 0)^T$  with all zeros except for one (the true class)):

$$p(y = c) = \prod_{c=1}^C \mu_c^{y_c}$$

Now, let  $p_{model}(y|x, w) = \prod_{c=1}^C f_w^{(c)}(x)^{y_c}$  be a **categorical distribution**. From probability theory, we know that maximizing this (log-) likelihood leads to minimizing the **cross-entropy (CE) loss**:

$$\begin{aligned} \hat{w}_{ML} &= \arg \max_w \sum_{i=1}^N \log p_{model}(y_i|x_i, w) = \arg \max_w \sum_{i=1}^N \log \prod_{c=1}^C f_w^{(c)}(x_i)^{y_{i,c}} = \\ &= \arg \max_w \underbrace{\sum_{i=1}^N \sum_{c=1}^C -y_{i,c} \log f_w^{(c)}(x_i)}_{CE \ Loss} \end{aligned}$$

Finally, we need to ensure that  $f_w^{(c)}(x)$  predicts a **valid categorical distribution**. In particular, we must guarantee that:

1.  $f_w^{(c)}(x) \in [0,1]$
2.  $\sum_{c=1}^C f_w^{(c)}(x) = 1$

The **softmax** function guarantees both of these properties:

$$\text{softmax}(x) = \left( \frac{\exp(x_1)}{\sum_{k=1}^C \exp(x_k)}, \dots, \frac{\exp(x_C)}{\sum_{k=1}^C \exp(x_k)} \right)$$

Let the score vector  $s$  denote the network output after the last affine layer. Then:

$$f_w^{(c)}(x) = \frac{\exp(s_c)}{\sum_{k=1}^C \exp(s_k)} \Rightarrow \log f_w^{(c)}(x) = s_c - \log \sum_{k=1}^C \exp(s_k)$$

Let us build intuition for the **log-softmax formulation**. Assume  $c$  is the correct class. Then our goal is to **maximize** the above mentioned criterion  $\log f_w^{(c)}(x)$ . The first term encourages the score  $s_c$  for the correct class  $c$  to increase. On the other hand, the second term (which can be approximated by:  $\log \sum_{k=1}^C \exp(s_k) \approx \max_k s_k$  as  $\exp(s_k)$  is insignificant for all  $s_k < \max_k s_k$ ) penalizes the most confident predictions. So, if the correct class already has the largest score (i.e.,  $s_k = \max_k s_k$ ), both terms roughly cancel and the example will contribute little to the overall

training cost. On the other hand, if the correct class doesn't have the largest score, then we incur loss.

It is worth noting that softmax only responds to differences between the inputs, hence it is invariant to adding the same scalar to all of the inputs:

$$\text{softmax}(x) = \text{softmax}(x + c)$$

We can therefore derive a numerically more stable variant:

$$\text{softmax}(x) = \text{softmax}\left(x - \max_{k=1,\dots,L} x_k\right)$$

which allows for accurate computation even when  $x$  is large.

Let us observe the behavior of the softmax function and the Cross-Entropy loss through a simple example:

Input $\mathbf{x}$	Label $\mathbf{y}$	Predicted scores $\mathbf{s}$	$\text{softmax}(\mathbf{s})$	CE Loss
	$(1, 0, 0, 0)^T$	$(+3, +1, -1, -1)^T$	$(0.85, 0.12, 0.02, 0.02)^T$	0.16
	$(0, 1, 0, 0)^T$	$(+3, +3, +1, +0)^T$	$(0.46, 0.46, 0.06, 0.02)^T$	0.78
	$(0, 0, 1, 0)^T$	$(+1, +1, +1, +1)^T$	$(0.25, 0.25, 0.25, 0.25)^T$	1.38
	$(0, 0, 0, 1)^T$	$(+3, +2, +3, -1)^T$	$(0.42, 0.16, 0.42, 0.01)^T$	4.87

Observe that for the first sample, the classifier is most confident in the correct label, thus we incur only a small loss. On the other hand, for the fourth sample, the classifier is least confident in the correct label, thus incurring a very high loss.

## OPTIMIZATION

Optimization is the general **process of identifying the combination of inputs to achieve the best possible output under certain constraints and conditions**. Several practical problems require an optimization technique to solve, like the traveling salesman problem for instance.

The objective here is to find the shortest and most efficient route for a person to take given a list of destinations.

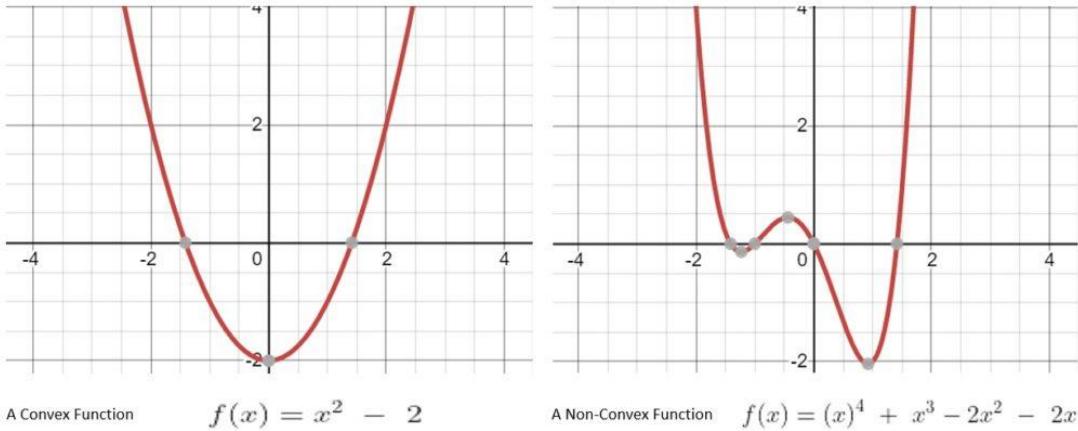
Perhaps an easier way to understand optimization is using a simple mathematical formulation like a linear function:

$$\begin{aligned} P(x, y) &= 4x + 3y \\ 2x + 4y &\leq 220, \quad 3x + 2y \leq 150, \quad x \geq 0, \quad y \geq 0 \end{aligned}$$

This function is known as the objective function in [linear programming](#). The idea is that it defines some quantity that needs to be maximized or minimized under the given constraints. There are other classes of optimization problems that involve higher-order functions, like [quadratic programming](#).

There are several ways to solve optimization problems, like linear programming or quadratic programming; however, it's imperative to understand that an important feature of these functions is that they are convex functions.

A [convex function](#) is one where if we draw a line from  $(x, f(x))$ , to  $(y, f(y))$ , then the graph of the convex function lies below that line. A function that doesn't exhibit this property is known as a non-convex function:



It's fairly intuitive to understand why this is important in optimization problems. A convex function can have only one optimal solution that is globally optimal. This makes solving such a problem much easier comparatively.

On the other hand, a non-convex function may have multiple locally optimal solutions. This makes it incredibly difficult to find the globally optimal solution.

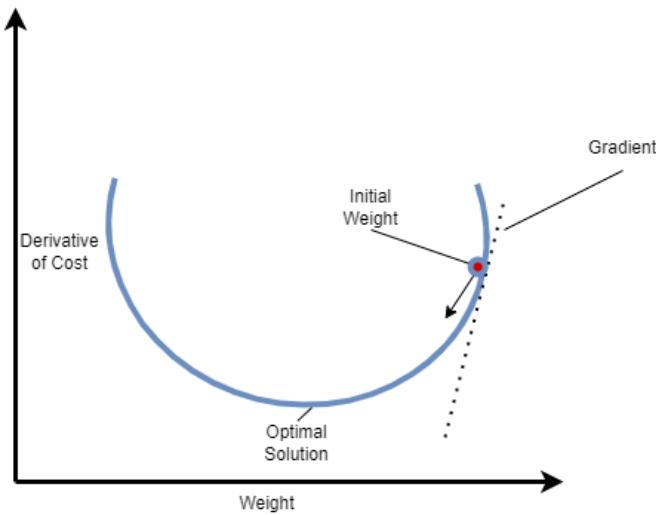
We'll talk about [Gradient Descent](#) (GD) and the different approaches that it includes. **Gradient Descent is a widely used high-level machine learning algorithm that is used to find a global minimum of a given function in order to fit the [training](#) data as efficiently as possible.** There are mainly three different types of gradient descent, **Stochastic Gradient Descent (SGD), Gradient Descent, and Mini Batch Gradient Descent.**

### *Gradient Descent*

**Gradient Descent is a widely used iterative optimization algorithm that is used to find the minimum of any differentiable function.** Each step that is taken improves the solution by proceeding to the negative of the gradient of the function to be minimized at the current point. In each iteration, the empirical cost  $E$  of a function is calculated and the parameters used to reach the global minimum are updated accordingly. **In order to find the optimal solution, this method takes into consideration the complete training dataset through every iteration, which may become time-consuming and computationally expensive.**

A common problem, in the way of finding the global minima, is that GD may converge to a local minimum and may not be able to get away from it; thus, failing to approach the minimum of a function.

We can see how Gradient Descent works in the case of a one-dimensional variable in the image below:



The formula of Gradient Descent that updates the weight parameter  $w$  is:

$$w_{i+1} = w_i - \alpha \nabla_{w_i} J(w_i)$$

Where  $w$  denotes the weights that need to be updated,  $\alpha$  is the Learning Rate of the algorithm, a hyperparameter that controls how much to change the model in response to the estimated error,  $J$  is the cost function, while  $i$  refers to the iteration index.

### *Stochastic Gradient Descent*

Stochastic Gradient Descent is a drastic simplification of GD which overcomes some of its difficulties. **Each iteration of SGD computes the gradient on the basis of one randomly chosen partition of the dataset which was shuffled, instead of using the whole part of the observations. This modification of GD can reduce the computational time significantly.** Nevertheless, this approach may lead to noisier results than the GD, because it iterates one observation at a time.

The formula of Stochastic Gradient Descent that updates the weight parameter  $w$  is:

$$w_{i+1} = w_i - \alpha \nabla_{w_i} J(x^i, y^i; w_i)$$

The notations are the same with Gradient Descent while  $y$  is the target and  $x$  denotes a single observation in this case.

### *Mini Batch Gradient Descent*

**Mini Batch Gradient Descent is considered to be the cross-over between GD and SGD. In this approach instead of iterating through the entire dataset or one observation, we split the dataset into small subsets (batches) and compute the gradients for each batch.**

The formula of Mini Batch Gradient Descent that updates the weight parameter  $w$  is:

$$w_{i+1} = w_i - \alpha \nabla_{w_i} J(x^{i:i+b}, y^{i:i+b}; w_i)$$

The notations are the same with Stochastic Gradient Descent where  $b$  is hyperparameter that denotes the size of a single batch.

### A Mathematical Approach

For a more algorithmic representation, let's assume the scenario below:

- A Hypothesis:  $h_w(x) = w_0 + w_1 x$
- Parameters that need to be updated:  $w_0, w_1$
- Cost Function:  $J(w_0, w_1) = \frac{1}{2m} \sum_{i=1}^m (h_w(x^{(i)}) - y^{(i)})^2$
- Aim: Minimization of  $J(w_0, w_1)$  with respect to  $w_0, w_1$

In the above hypothesis, the computation of  $w$  differs for every approach. **The three different methods calculate  $w$  until it converges accordingly.**

Pseudocode for Gradient Descent

---

#### **Algorithm 1:** Pseudo-code for GD

---

##### **Function GD:**

```

Set epsilon as the limit of convergence
for  $j = 1$  and  $j = 0$  do
    while  $|\omega_{j+1} - \omega_j| < \text{epsilon}$  do
         $\omega_{j+1} := \omega_j - \alpha \cdot \frac{1}{m} \sum_{i=1}^m (h_\omega(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$ ;
    end
end

```

---

$m$  is the number of training samples. We can see that, depending on the dataset, Gradient Descent may have to iterate through many samples, which can lead to being unproductive. **To calculate the gradient of the cost function**, we need to sum the cost of each sample. If we have 3 million samples, we have to loop through 3 million times or use the dot product.

Code example:

```

def gradientDescent(X, y, theta, alpha, num_iters):
    """
        Performs gradient descent to learn theta
    """
    m = y.size # number of training examples
    for i in range(num_iters):
        y_hat = np.dot(X, theta)
        theta = theta - alpha * (1.0/m) * np.dot(X.T, y_hat-y)
    return theta

```

Do you see `np.dot(X.T, y_hat-y)` above? That's the vectorized version of "looping through (summing) 3 million samples".

So to move a single step towards the minimum, do we really have to calculate each cost 3 million times?

Yes. If you insist to use GD.

But if you use Stochastic GD, you don't have to!

Pseudocode for Stochastic Gradient Descent

---

**Algorithm 2:** Pseudo-code for SGD

---

**Function SGD:**

```

Set epsilon as the limit of convergence
for  $i = 1, \dots, m$  do
    for  $j = 0, \dots, n$  do
        while  $|\omega_{j+1} - \omega_j| < \text{epsilon}$  do
             $\omega_{j+1} := \omega_j - \alpha \cdot (h_\omega(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)};$ 
        end
    end
end

```

---

As we can see, in this case, the gradients are calculated on one random shuffled part out of  $n$  partitions.

```

def SGD(f, theta0, alpha, num_iters):
    """
    Arguments:
        f -- the function to optimize, it takes a single argument
             and yield two outputs, a cost and the gradient
             with respect to the arguments
        theta0 -- the initial point to start SGD from
        num_iters -- total iterations to run SGD for           Return:
        theta -- the parameter value after SGD finishes
    """
    start_iter = 0
    theta= theta0    for iter in xrange(start_iter + 1, num_iters + 1):
        _, grad = f(theta)
        theta = theta - (alpha * grad) # there is NO dot product!
    return theta

```

Pseudocode for Mini Batch Gradient Descent

Let's assume  $b$  batches.

---

**Algorithm 3:** Pseudo-code for Mini Batch Gradient Descent

---

**Function SGD:**

```

    Set epsilon as the limit of convergence
    for  $i = 1, \dots, b$  do
        for  $j = 0, \dots, n$  do
            while  $|\omega_{j+1} - \omega_j| < epsilon$  do
                 $\omega_{j+1} := \omega_j - \alpha \cdot \frac{1}{b} \sum_{k=i}^{i+b-1} (h_\omega(x^{(k)}) - y^{(k)}) \cdot x_j^{(k)}$ ;
            end
        end
    end

```

---

In this case, the gradients are calculated with subsets of all observations in each iteration. This algorithm converges faster and concludes in more precise results.

#### *Comparison of Gradient and Stochastic Gradient Descent*

It is important to make a head-to-head comparison between the two main approaches, by analyzing the advantages and disadvantages of each one.

Gradient Descent:

Advantages	Disadvantages
Noisless and lower standard error	Time consuming
Unbiased estimation of the gradients	Computationally expensive
The path for finding the global minima is guaranteed to converge	

Stochastic Gradient Descent:

Advantages	Disadvantages
Generally faster	Noisier results
Lighter computationally	Overfits and results in large variance and small bias
Randomization causes model generalization	

**Mini Batch Gradient Descent is the bridge between the two approaches above.** By taking a subset of data we result in fewer iterations than SGD, and the computational burden is also reduced compared to GD. This middle technique is usually more preferred and used in machine learning applications.

## REGULARIZATION – PREVENT OVERFITTING

**Regularization** is a technique in machine learning that tries to **achieve the generalization of the model**. It means that our model works well not only with training or test data, but also with the data it'll receive in the future. In summary, to achieve this, regularization shrinks the weights toward zero to discourage complex models. Accordingly, this avoids overfitting and reduces the variance of the model.

**There are three main techniques for regularization in linear regression:**

1. Lasso Regression
2. Ridge Regression
3. Elastic Net

### *Lasso Regression*

Lasso regression, or L1 regularization, is a technique that increases the cost function by a penalty equal to the sum of the absolute values of the non-intercept weights from linear regression.

Formally, we define L1 regularization as:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m (\theta^T \cdot x^{(i)} - y^{(i)})^2 + \lambda \sum_{j=1}^n |\theta_j|$$

Here,  $\lambda$  is a regularization parameter. Basically,  $\lambda$  controls the degree of regularization. In particular, the higher the  $\lambda$  is, the smaller the weights become. **In order to find the best  $\lambda$ , we can start with  $\lambda = 0$  and measure the cross-validation error at each iteration, increasing the  $\lambda$  with a fixed value.**

### *Ridge Regression*

Similar to Lasso Regression, Ridge Regression, or L2 regularization, adds a penalty to the cost function. The only difference is that the penalty is calculated using the squared values of non-intercept weights from linear regression. Thus, we define L2 regularization as:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m (\theta^T \cdot x^{(i)} - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2$$

**Like with L1 regularization, we can estimate the  $\lambda$  parameter in the same way. The only difference between Lasso and Ridge is that Ridge converges faster, while Lasso is more commonly used for feature selection.**

### *Elastic Net Regression*

**In summary, this method is a combination of the previous two approaches; it combines penalties from both Ridge and Lasso Regression.** Therefore, we define [Elastic Net](#) regularization with the formula:

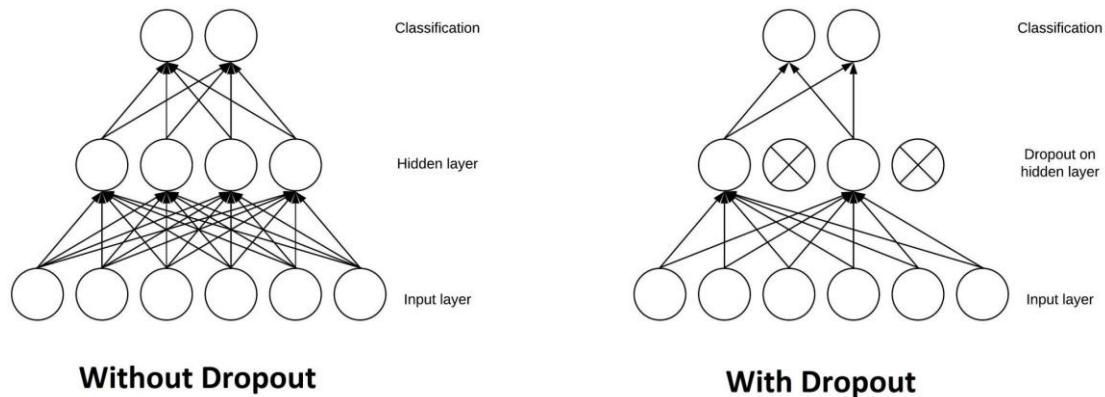
$$J(\theta) = \frac{1}{m} \sum_{i=1}^m (\theta^T \cdot x^{(i)} - y^{(i)})^2 + \lambda_1 \sum_{j=1}^n |\theta_j| + \lambda_2 \sum_{j=1}^n \theta_j^2$$

In contrast to the previous two methods, here we have two regularization parameters,  $\lambda_1$  and  $\lambda_2$ . As such, we'll have to find both of them.

### *Dropout Regularization*

The **Dropout layer** is a mask that **nullifies the contribution of some neurons towards the next layer and leaves unmodified all others**. We can apply a Dropout layer to the input vector, in which case it nullifies some of its features; but we can also apply it to a hidden layer, in which case it nullifies some hidden neurons.

Dropout layers are important in training CNNs because they prevent overfitting on the training data. If they aren't present, the first batch of training samples influences the learning in a disproportionately high manner. This, in turn, would prevent the learning of features that appear only in later samples or batches:



Say we show ten pictures of a circle, in succession, to a CNN during training. The CNN won't learn that straight lines exist; as a consequence, it'll be pretty confused if we later show it a picture of a square. We can prevent these cases by adding Dropout layers to the network's architecture, in order to prevent overfitting.

### *Data Augmentation*

**Data augmentation** aims to **improve the quality of our dataset** when we don't have hundreds of thousands of samples.

#### Geometric Transformations

The simplest techniques for image data augmentation apply **geometric transformations**. These include **flipping, scaling, cropping, rotation, translation, and brightness transformations**:



We should consider applying this type of data augmentation when we don't have a robust sample distribution. For example, if we want to recognize dogs in pictures taken by other users in an app. Most likely, these users will take pictures in different orientations and lighting conditions.

If our dataset consists of samples collected under similar scenarios, we won't be able to recognize these different samples. To keep it short, we should use geometric transformations when we have enough unique samples that are collected without many variations.

Whichever transformation we use, we should **make sure that our application isn't sensitive to it**. For instance, if we vertically flip the number 9, the image will become a 6, so the initial label will be wrong. For this reason, we usually **need domain-specific functions and methods to augment our dataset**.

Where to add the data?

The most common practice is to apply data augmentation only to the training samples. The reason is that we want to increase our model's generalization performance by adding more data and diversifying the training dataset. However, we can also use it during testing.

In all cases, we should keep in mind that the augmentation parameters and strategy have to be appropriate to the application. This is true no matter in which phase we augment the data. The choice of data augmentation strategy should be application-specific and in line with the real-world data distribution.

*Should we augment test data?*

An elementary reason to not use data augmentation on the test data is to **have a realistic evaluation of the model**. To check how our model behaves on unseen data, test samples should come from the same distribution as those we expect in our real-world application. **Data augmentation techniques may distort the test data distribution and skew the results**.

Still, we can find in the literature some evidence that argues for the use of augmentation on the test set. In fact, the method called [Test-Time Augmentation \(TTA\)](#) improved the accuracy in certain applications.

For instance, let's suppose our task is to develop a model for facial recognition regardless of the image orientation. But, our dataset is quite small with just a few hundred samples. Also, all the photos were taken in only one position and direction.

We can carry out data augmentation only in the training phase. But, in that case, we wouldn't accurately assess our model for our target application, which can include rotated images.

## Why it works

Convolutional Neural Networks are very efficient because:

1. They are invariant to geometrical transformations and learn features that get increasingly complicated and detailed, hence being powerful hierarchical **feature extractors** thanks to the convolutional layers.
2. They combine the extracted features and aggregate them in a nonlinear fashion to predict the output and therefore being **robust classifiers** thanks to the fully connected layers.

## TRANSFER LEARNING

**Transfer learning is a very popular technique in deep learning where existing [pre-trained](#) models are used for new tasks. Basically, the idea is to take one neural network that we trained for one specific task and use it as a starting point for another task.** Sometimes, when we need to use an existing neural network for a different task, for example, the classification of three classes instead of two, it is possible to change only a few last layers of the network and to keep the rest as it is.

Nowadays, in computer vision and natural language processing fields, researchers and engineers use huge neural networks that require vast computing and time resources to learn. Instead of retraining these models every time, thanks to transfer learning, we can take these models as starting points for our tasks and fine-tune them a bit for our purpose. Of course, the more similar the original training data is to our data set, the less fine-tuning we need to do.

We're going to explore the differences and nuances of **transfer learning** and **domain adaptation**. **Transfer learning is a broad term that describes using the knowledge gained from one machine learning problem in another one. Domain adaptation describes a special case of transfer learning that only covers the change of the data domain.**

### Classical Supervised Learning

To clarify transfer learning, we compare it to a classical [supervised machine learning](#) problem.

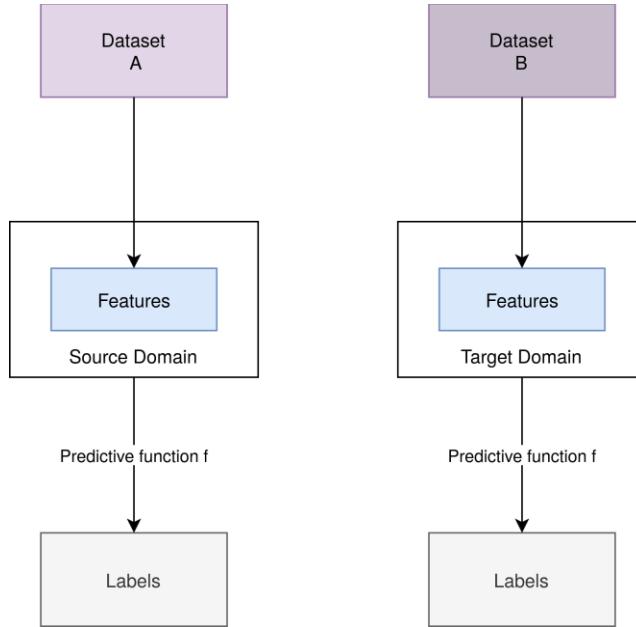
For a better understanding, we work with a sample dataset of dog and cat pictures.

#### STRUCTURE

In a classical setting, we have a dataset  $A$  form which we extract our features  $x$ . For those features, we want labels  $y$ , for example, a picture that has the high-level features, foot of a dog, the face of a dog, and body of the dog, we assign the label dog. Thus we have a function assigning values of our set features to our set labels.

**When we use the model in production, we have a different but similar dataset, it still shows dogs and cats.** And it still assigns those pictures to the same labels “dog” and “cat”.

In this picture, we can see two datasets  $A$  and  $B$ , which are different but similar:



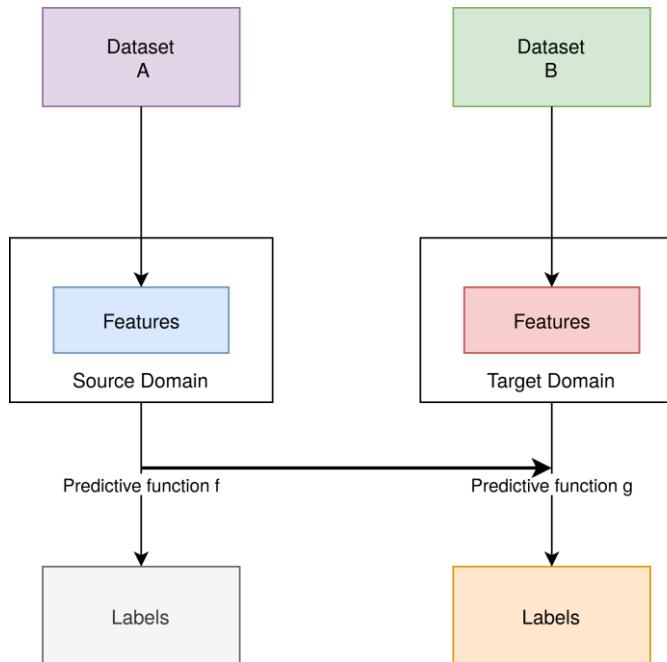
From the datasets, we extract our features. And for a particular set of features, we assign one or more labels. The features, the predictive function  $f$ , and the labels remain the same.

## Transfer Learning

**Transfer Learning describes a collection of machine learning techniques that work with a structure similar to the classical supervised case.** In contrast, it can also work with datasets and features that are vastly different.

### TRANSFER LEARNING BLUEPRINT

Let's have a look at the structure of a transfer learning process. As we can see, the labels and the predictive function can change:



Furthermore, we have a similar structure to supervised learning. But, in heavy contrast, none of the building blocks have to be the same in the case of transfer learning. **The connection between the two machine learning settings is the utilization of the predictive function  $f$ , which is used in the creation of the second predictive function  $g$ .**

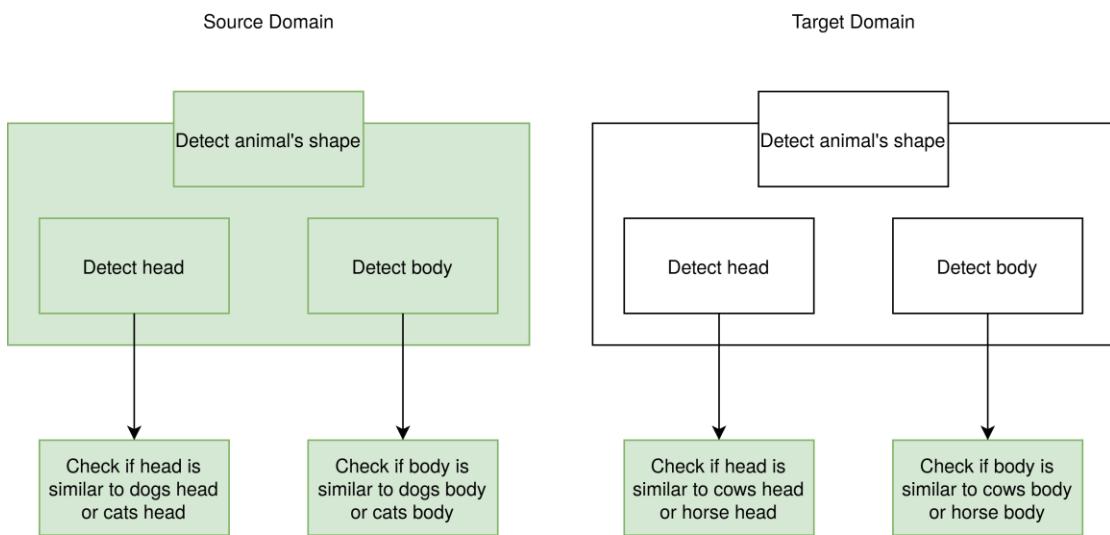
It should be noted that in this case, the steps of source and target are different. Transfer learning also includes the other cases, in which e.g. only labels are different and features are the same.

The instance that covers the case of the same labels and vastly different, but similar datasets is a type of transfer learning as well. We will cover this in the section about domain adaptation.

### *Transfer Learning Example*

Let's apply this concept to our example of dog and cat pictures. Now imagine we have a second dataset that shows pictures of cows and horses. Cows and horses are significantly different from cats.

Nevertheless, they are all mammals, they have four feet, and a similar shape. As a solution, we can take the layers that describe the shape of the object we want to detect, whether it is a dog, a cow, or a horse, and freeze them. **Freezing means we cut them out of our predictive function  $f$  put them in our predictive function  $g$ , and train the function  $g$ , without training our frozen layers:**



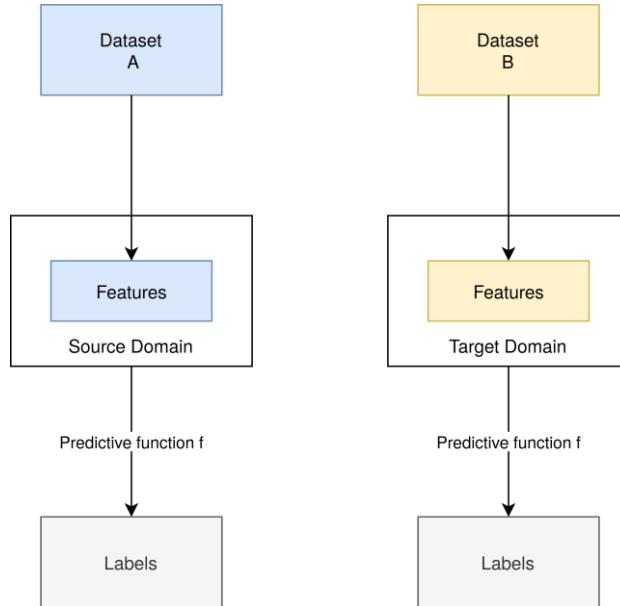
We can see the layers on the right side are green, which indicates that we have to train them while creating our predictive function for the source domain. On the other hand, the predictive  $g$  is created using the already existing, frozen layer from the source domain. The frozen layers stay untouched during the training process.

## DOMAIN ADAPTATION

Domain adaptation is, as already mentioned, a special case of transfer learning.

### The Blueprint

In domain adaptation, we solely change the underlying datasets and thus the features of our machine learning model. **However, the feature space stays the same.** The predictive function  $f$  stays the same:



### Application

Applying domain adaptation to our example, we could think of a significantly different, but somehow similar dataset. **This could still contain dog and cat pictures, but those that are vastly different from the ones in our source dataset.** For example, in our source data set, we only have poodles and black cats. In our target dataset, on the other hand, we could have schnauzers and white cats.

Now, how can we ensure that our predictive function will still predict the right labels for our dataset? Domain adaptation delivers an answer for this question.

### Types of Domain Adaptation

We consider three types of domain adaptation. These are defined by the number of labeled examples in the underlying domain:

- **Unsupervised domain adaptation** works with a source domain that has labeled examples, but also unlabeled examples. The target domain only has unlabeled examples.
- **Semi-supervised domain adaptation** expects some of the examples in the target domain are labeled.
- **Supervised domain** indicates that all examples are labeled.

## METHODS IN DOMAIN ADAPTATION

In domain adaptation, we can look a bit closer at pragmatic approaches. **This lies in the fact, that only changing the dataset makes it much easier to tune our model for our new machine learning process.**

### *Divergence-based Domain Adaptation*

Divergence-based domain adaptation is a method of testing if two samples are from the same distribution. As we have seen in our blueprint illustration, the features that are extracted from the datasets are vastly different. This difference causes our predictive function to not work as intended. If it's fed by features that it was not trained for, it malfunctions. This is also the reason why we accept different features but require the same feature space.

**For this reason, divergence-based domain adaptation creates features that are “equally close” to both datasets.** This can be achieved by applying various algorithms, including the Maximum Mean Discrepancy, Correlation Alignment, Contrastive Domain Discrepancy, or the Wasserstein Metric.

### *Iterative Approach*

In the iterative approach, we use our prediction function  $f$  to label those samples of our target domain, for which we have very high confidence. Doing so, we retrain our function  $f$ . **Thus creating a prediction function that fits our target domain more and more as we apply it to samples that have less confidence.**

## Why Does Transfer Learning Work?

Transfer learning is possible because neural networks tend to learn different patterns in the data. For example, in computer vision, initial layers learn low-level features such as lines, dots, and curves. Top layers learn high-level features built on top of low-level features. Most of the patterns, especially low-level ones, are common to many different computer image data sets. Instead of learning them every time from scratch, we can use existing parts of the network and reuse them with a bit of tweaking for our purposes.

## RECOGNITION

Object recognition is a generic term to indicate a set of computer vision tasks for identifying objects in digital images. We can distinguish several tasks that belong to the object recognition field: image classification, object localization, object detection, semantic segmentation, and instance segmentation. Nowadays, machine learning and deep learning techniques are the best way to perform object recognition tasks.

### Object Recognition Tasks

Let's now analyze in detail the main object recognition tasks.

#### IMAGE CLASSIFICATION

**Image classification deals with assigning a single label to an image.** Hence, an image classification system takes as **input an image with one or more objects** and **returns a single label**. For example, given an image depicting a room, an image classification model would return one of the following labels: "kitchen", "bathroom", "living room".



Now that we have defined what image classification is, let us provide further motivation. First of all, it is an extremely useful task on its own (e.g. it allows for effective image search from a large database). Furthermore, it is a very **standardized task** - it embodies simple input/output specification (e.g. inputs are 224x224 images, and outputs are singular labels such as "cat", "dog", etc.) and can be trained using standard loss functions (such as categorical cross entropy). In fact, for a long time this has been one of the difficult gold-standard tasks in computer vision. Finally, it is a very useful high-level proxy-task for learning good representations which can be transferred to a completely new task/domain (this process is also known as **transfer learning**).

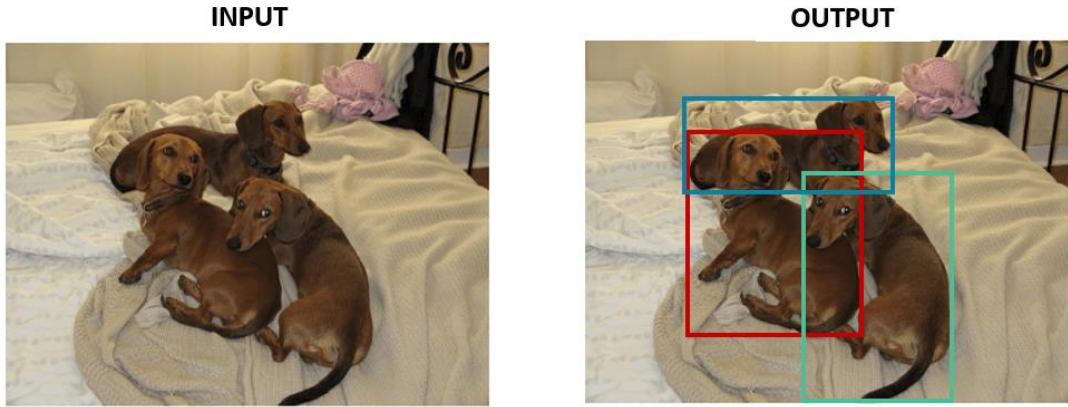
Convolutional Neural Networks (CNNs) are the most performing solution for image classification. The most common CNN architectures are AlexNet, VGG-16, Inception-v3, Xception, and EfficientNet.

#### OBJECT LOCALIZATION

**Object localization deals with localizing objects by indicating their bounding boxes.** Hence, an object localization algorithm takes as **input an image** and **returns a set of bounding boxes, each**

**one representing the location of a detected object.** The bounding box is generally defined by a point (the center or the top-left coordinate), the width and the height of the rectangle. An object localization system doesn't return any label related to the category of the object.

An example of object localization is shown in the following figure:



### *Localization Approaches*

#### Sliding Window

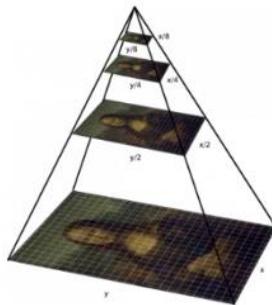
We take windows of fixed sizes from input image at all the possible locations feed these patches to an image classifier. But there is a problem, how would we know the size of the window so that it always contains the image?

#### Image Pyramid

**Image pyramid** is created by scaling the image, where the idea is to resize the image at multiple scales and rely on the fact that our chosen window size will completely contain the object in one of these resized images. Most commonly, the image is downsampled (size is reduced) until a certain condition, typically a minimum size, is reached.

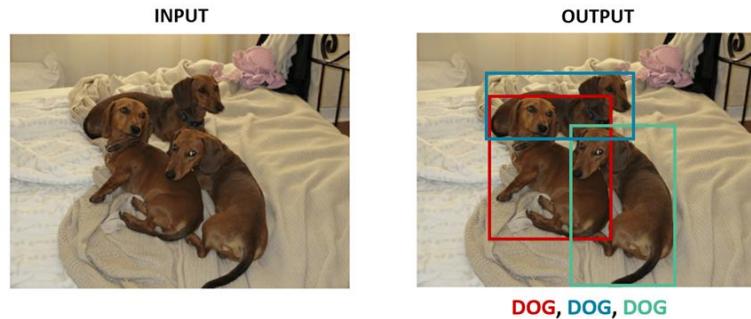
A fixed size window detector is run on each of these images and it's common to have as many as 64 levels on such pyramids. Now, all these windows are fed to a classifier to detect the object of interest.

This approach can be very expensive computationally, and thus very slow.



## OBJECT DETECTION

**Object detection is the task of locating objects in an image with bounding boxes and assigning a class label for each detection.** Hence, an object detection system takes as **input an image**, with one or more objects, and **returns a set of bounding boxes and the corresponding class labels**:



We can say that:

$$\text{Object localization} + \text{Image Classification} = \text{Object Detection}$$

Nowadays, the most performing CNN architectures for object detection are Faster R-CNN, YOLO, EfficientDet and Single Shot Detector (SSD).

## What matters in recognition?

Learning Techniques	Representation	Data
<ul style="list-style-type: none"> <li>▸ E.g. choice of classifier or inference method</li> </ul>	<ul style="list-style-type: none"> <li>▸ <i>Low level</i>: SIFT, HoG, GIST, edges, etc.</li> <li>▸ <i>Mid level</i>: Bag of words, sliding window, deformable model</li> <li>▸ <i>High level</i>: Contextual dependence</li> <li>▸ Deep learned features</li> </ul>	<ul style="list-style-type: none"> <li>▸ More is always better (as long as it is good data) ??</li> <li>▸ Annotation is hard</li> <li>▸ Often even collection is hard</li> </ul>

## What's still hard?

### FINE-GRAIN CLASSIFICATION

The Fine-Grained Image Classification task focuses on differentiating between hard-to-distinguish object classes, such as species of birds, flowers, or animals; and identifying the makes or models of vehicles.

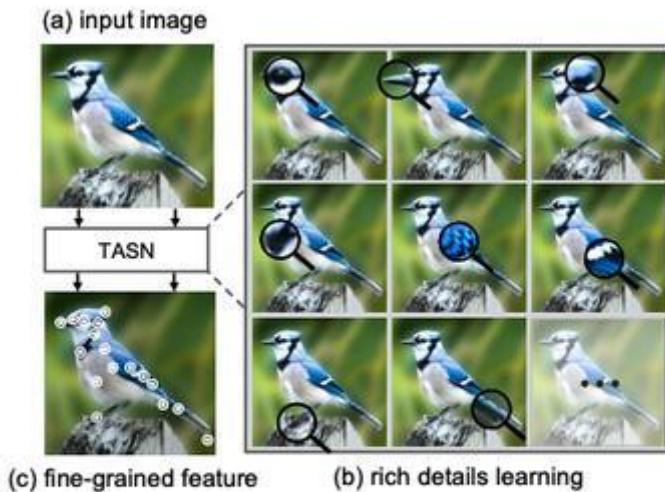


Figure 1. An illustration of learning discriminative details by TASN for a “blue jay.” As shown in (b), TASN learns such subtle details by up-sampling each detail into high-resolution. And the white concentric circles in (c) indicates fine-grained details.

### FEW-SHOT LEARNING (FSL)

Few-shot learning (FSL), also referred to as low-shot learning (LSL) in few sources, is a type of machine learning method where the training dataset contains limited information.

The common practice for machine learning applications is to feed as much data as the model can take. This is because in most machine learning applications feeding more data enables the model to predict better. However, few shot learning aims to build accurate machine learning models with less training data.

Few-shot learning algorithms coupled with a data-centric approach to model development can help companies reduce data analysis/machine learning (ML) costs since the amount of input data is an important factor that determines resource costs (e.g., time and computation).

## IMAGE SEGMENTATION

**There are two main types of segmentation: instance segmentation and semantic segmentation.**

### Semantic Segmentation

In **semantic segmentation**, all the objects that belong to the same class share the label. So, if we're working with autonomous vehicle applications, all pedestrians will receive the same label. The same goes for cars. For instance:



The main deep learning models for semantic segmentation are Fully Convolutional Network (FCN), U-Net and DeepLab.

### Instance Segmentation

In instance segmentation, each detected object receives its unique label. **We usually implement this type of segmentation when the number of objects or their independence is relevant.** For instance, we may want to count people at a concert. To do so, we need to isolate and differentiate each visitor.

Returning to our example with an autonomous vehicle, each pedestrian and car will receive unique labels (which we represent using different colors):



The most important deep learning models for instance segmentation are Mask R-CNN, MaskLab, and TensorMask.

**In image-segmentation tasks, we identify each pixel's class.**

**A group of pixels belonging to the same class constitutes a segment.** Usually, we aim to create segments by isolating objects in an image. By doing that, we change the image's representation into a new one. Instead of actual pixel values, pixels in a segmented image can be thought of as containing class labels.

There are several methods of doing that, and the simplest one is thresholding. If we have a [gray-scale](#) image in which we want to split the foreground from the background, we define a threshold, let's say 149. All the pixels below this value will be considered to belong to the foreground. In contrast, the background will consist of every pixel with a value above 149. For example:



Input Image



Segmented Image

Biology gives us one of the oldest applications of image segmentation. If we have a sample from human tissue, we might need to isolate cancer cells from healthy ones. Segmentation is warranted in this case because the structure of a cancer cell differs from that of a normal cell, so an image would reflect that.

So, after we isolate the cells with segmentation tools, we can continue with the morphological analysis.

## Segmentation as Clustering

As we have shown, we can map an image to a feature space, and in that space we can group (cluster) together points that have similar features. This visual similarity can be based on:

- Brightness
- Color
- Position
- Depth
- Motion
- Etc.

In order to perform segmentation or clustering, we need some definition for the **similarity between pixels**.

### PIXEL SIMILARITY

Let  $i$  and  $j$  be two pixels whose features are  $f_i$  and  $f_j$ .

**$L^2$  distance** between  $f_i$  and  $f_j$ :

$$S(f_i, f_j) = \sqrt{\sum_k (f_{ik} - f_{jk})^2}$$

**Smaller the Distance – Greater Similarity.**

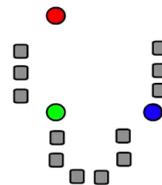
### K-MEANS SEGMENTATION

We want to segment the given pixel feature distribution into  $k$  clusters. We will show the algorithm for  $k = 3$  clusters.

*Steps:*

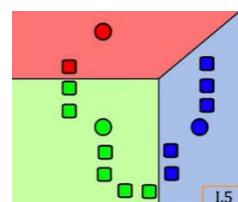
1. Randomly generate the initial centroids (**means**) of the 3 clusters.

Generally: Pick  $k$  points randomly as the initial centroids (means)  $\{m_1, m_2, \dots, m_k\}$  of  $k$  clusters in feature space.



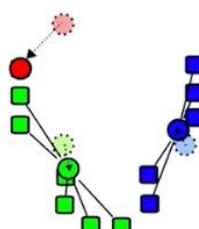
2. Create 3 clusters by assigning each feature point to the nearest mean.

Generally: For each pixel  $x_j$  find nearest cluster mean  $m_i$  to pixel's feature  $f_j$  and assign pixel to cluster  $i$ .



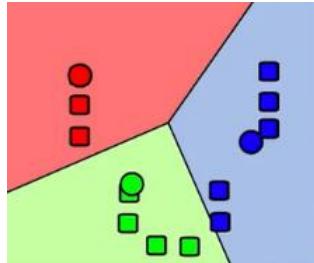
3. Recompute the mean of each cluster.

Generally: Recompute mean for each cluster using its assigned pixels.



4. Repeat steps 2 and 3 until convergence.

Generally: If changes in all  $k$  means is less than a threshold  $\varepsilon$ , stop. Else go to step 2.



Initialization methods

Method 1: Select  $k$  random feature points as initial centroids. If two points are very close, resample.

Method 2: Select  $k$  uniformly distributed means within the range of the distribution.

Method 3: Perform  $k$ -means clustering on a subset of pixels and use the result as the initial means.

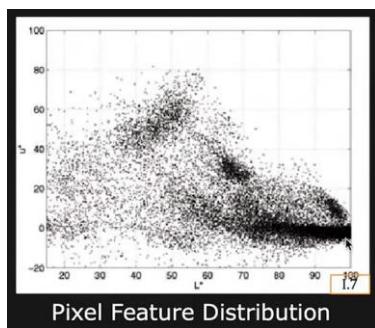
### *Conclusion*

This algorithm is:

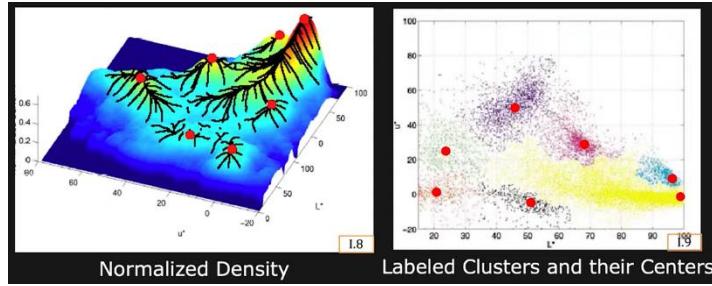
- Simple and reasonably fast
- **Need to pick the number of clusters  $k$ .**
- Sensitive to initialization
- Sensitive to outliers

## MEAN-SHIFT SEGMENTATION

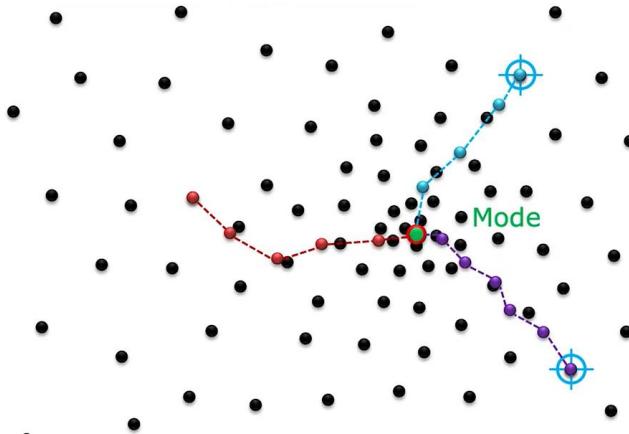
We can look at Pixel Feature Distribution:



And represent it as Normalized Density, where each hill represents a cluster. Peak (mode) of hill represents “center” of cluster. Each pixel climbs the steepest hill within its neighborhood, then the pixel is assigned to the hill (cluster) it climbs.



First, we want to find the hill (cluster) that the sample pixel belongs to. We will compute centroid within a window of size  $W$ . We then take all the points within the window and calculate the centroid. After this step we move to the mean (mean shift), and repeat the process until we stuck at the mode (the peak of this hill). The idea is that **features that converge to the same mode belong to same cluster.**



### *Algorithm*

Given a distribution of  $N$  pixels in feature space, we want to find modes (clusters) of distribution.

#### **Clustering:**

1. Set  $m_i = f_i$  as initial mean for each pixel  $i$ .
2. Repeat the following for each mean  $m_i$ :
  - a. Place window of size  $W$  around  $m_i$ .
  - b. Compute centroid  $m$  within the window. Set  $m_i = m$ .
  - c. Stop if shift in mean  $m_i$  is less than a threshold  $\varepsilon$ .  $m_i$  is the mode.
3. Label all pixels that have same mode as belonging to same cluster.

### *Conclusion*

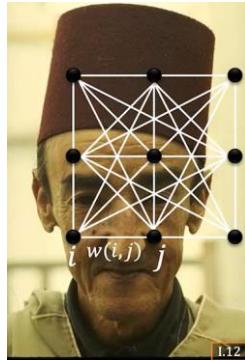
This algorithm is:

- Simple but computationally expensive
- Finds arbitrary number of clusters
- No initialization required
- Robust to outlier
- Clustering depends on window size  $W$

## Graph Based Segmentation

Images are represented as graphs, where:

- A vertex for each pixel.
- An edge between each pair of pixels.
- Graph Notation:  $G = (V, E)$  where  $V$  and  $E$  are the sets of vertices and edges, respectively.
- Each edge is weighted by the **affinity** or similarity between its two vertices.



### MEASURING AFFINITY

Let  $i$  and  $j$  be two pixels whose features are  $f_i$  and  $f_j$ .

**Pixel Dissimilarity** (L2 Norm between features):

$$S(f_i, f_j) = \sqrt{\sum_k (f_{ik} - f_{jk})^2}$$

**Pixel Affinity:**

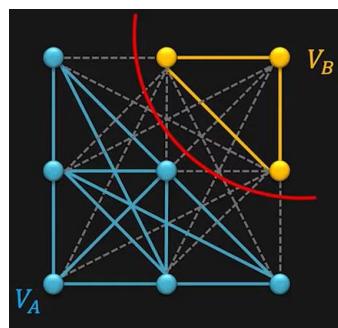
$$w(i, j) = A(f_i, f_j) = e^{-\frac{1}{2\sigma} S(f_i, f_j)}$$

**Smaller the Dissimilarity – Larger the Affinity.**

### GRAPH CUT

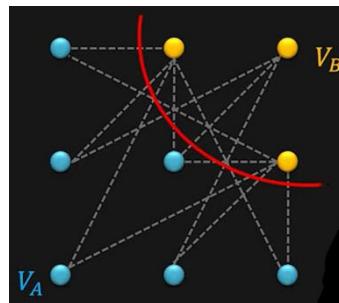
**Cut**  $C = (V_A, V_B)$  is a partition of vertices  $V$  of a graph  $G = (V, E)$  into two **disjoint subsets**  $V_A$  and  $V_B$ .

**Cut-Set:** Set of edges whose vertices are in different subsets of partition.



**Cost of Cut:** Sum of weights of cut-set edges.

$$\text{cut}(V_A, V_B) = \sum_{u \in V_A, v \in V_B} w(u, v)$$

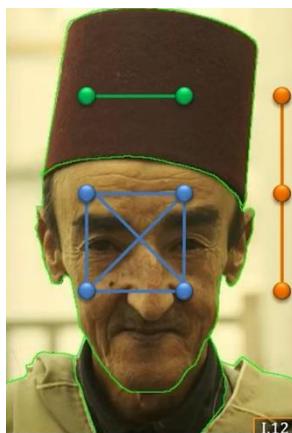


## GRAPH CUT SEGMENTATION

Criteria for Graph Cut:

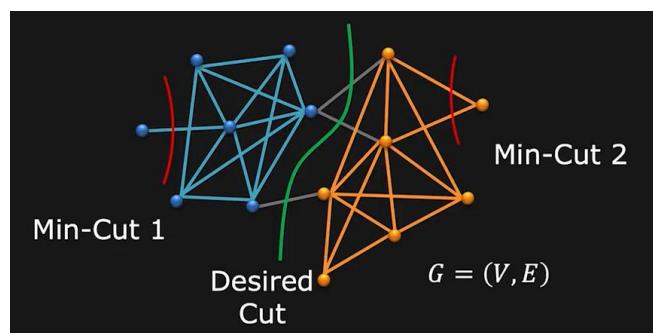
- A pair of vertices (pixels) within a subgraph has **high affinity**.
- A pair of vertices from two different subgraphs has **low affinity**.

That is, minimize the cost of cut. Also called **Min-Cut**. See each subgraph is an image segment.



### Problem with Min-Cut

There is a bias to cut small, isolated segments.



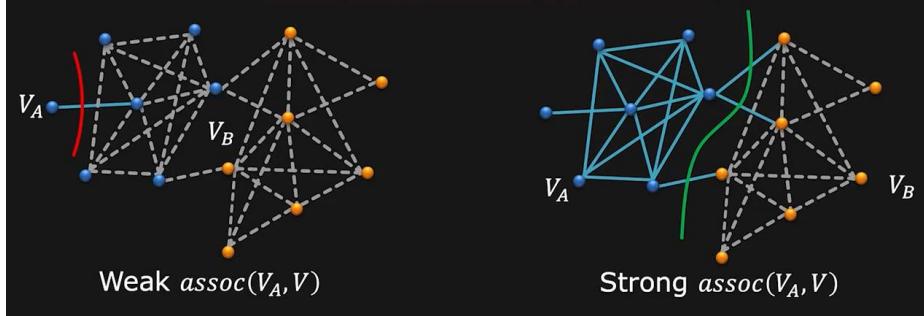
The desired cut costs more than Min-Cut 1 or Min-Cut 2.

**Solution:** Normalize Cut to favor larger subgraphs.

## MEASURE OF SUBGRAPH SIZE

Compute how strongly vertices  $V_A$  are associated with vertices  $V$ .

$$assoc(V_A, V) = \sum_{u \in V_A, v \in V} w(u, v)$$



$assoc(\cdot)$  is the sum of weights of the solid edges.

## NORMALIZED CUT (NCUT)

**Minimize Cost of Normalized Cut** during Partition:

$$NCut(V_A, V_B) = \frac{cut(V_A, V_B)}{assoc(V_A, V)} + \frac{cut(V_A, V_B)}{assoc(V_B, V)}$$

Minimizing NCut has no known polynomial time solution. It is **NP-Complete**.

## Conclusion

This algorithm is:

- Generic framework, can be used with many different features and affinity formulations
- No model or data distribution
- High storage requirement and time complexity
- Bias toward partitioning into equal segments

## MARKOV RANDOM FIELD (MRF)

### Potential

A **potential**  $\phi(x)$  is a non-negative function of the variable  $x$ . A **joint potential**  $\phi(x_1, x_2, \dots, x_D)$  is a non-negative function of the **set** of variables.

### Markov Random Field (MRF) = Markov Network

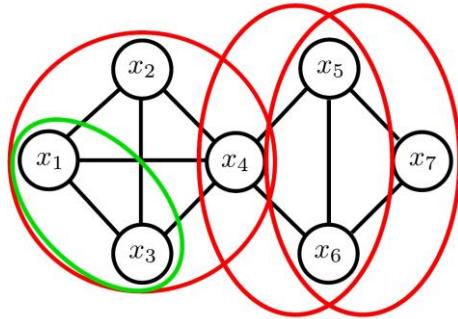
For a set of variables  $\chi = \{x_1, x_2, \dots, x_D\}$  a **Markov Random Field** is defined as a product of potentials over the **(maximal) cliques**  $\{\chi_c\}_{c=1}^C$  of the undirected graph  $G$ :

$$p(\chi) = \frac{1}{Z} \prod_{c=1}^C \phi_c(\chi_c)$$

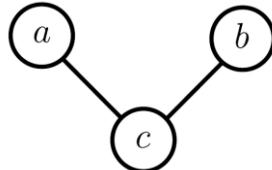
where  $Z$  is an appropriate normalization constant (also called the partition function),  $C$  are the (maximal) cliques of the graph.

### *Undirected Graph*

An **undirected graph**  $G$  is a graph with vertices and undirected edges. A **clique** (green, red) is a subset of vertices that are fully connected, and a **maximal clique** (red) is a clique that cannot be extended by any other vertex.



### *Properties of Markov Random Fields*



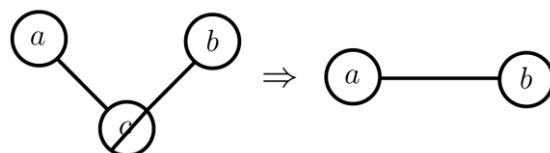
$$p(a, b, c) = \frac{1}{Z} \phi_1(a, c) \phi_2(b, c)$$

Two maximal cliques of size two:  $\phi_1(a, c), \phi_2(b, c)$

$Z$  normalizes the distribution and is called **partition function**.

$$Z = \sum_{a,b,c} \phi_1(a, c) \phi_2(b, c)$$

Marginalizing over  $c$  makes  $a$  and  $b$  dependent.



We can prove it by showing that  $a$  and  $b$  are not independent:

$$p(a, b) \neq p(a)p(b)$$

Let's show this statement by contradiction. Assume the following holds true:

$$\begin{aligned} p(a, b) &= \sum_c p(a, b, c) = \frac{1}{Z} \sum_c \phi_1(a, c) \phi_2(b, c) = \\ &= p(a)p(b) = \sum_{b,c} p(a, b, c) \sum_{a,c} p(a, b, c) = \frac{1}{Z} \sum_{b,c} \phi_1(a, c) \phi_2(b, c) \frac{1}{Z} \sum_{a,c} \phi_1(a, c) \phi_2(b, c) \end{aligned}$$

Therefore, we have:

$$\begin{aligned}
 \frac{1}{Z} \sum_c \phi_1(a, c) \phi_2(b, c) &= \frac{1}{Z} \sum_{b,c} \phi_1(a, c) \phi_2(b, c) \frac{1}{Z} \sum_{a,c} \phi_1(a, c) \phi_2(b, c) \Rightarrow \\
 \Rightarrow \sum_c \phi_1(a, c) \phi_2(b, c) &= \frac{1}{Z} \sum_{b,c} \phi_1(a, c) \phi_2(b, c) \sum_{a,c} \phi_1(a, c) \phi_2(b, c) \Rightarrow \\
 \Rightarrow Z \sum_c \phi_1(a, c) \phi_2(b, c) &= \sum_{b,c} \phi_1(a, c) \phi_2(b, c) \sum_{a,c} \phi_1(a, c) \phi_2(b, c) \Rightarrow \\
 \Rightarrow \sum_{a,b,c} \phi_1(a, c) \phi_2(b, c) \sum_c \phi_1(a, c) \phi_2(b, c) &= \sum_{b,c} \phi_1(a, c) \phi_2(b, c) \sum_{a,c} \phi_1(a, c) \phi_2(b, c)
 \end{aligned}$$

Consider the following example, where  $a, b, c$  are binary variables:

$$\phi_1(a, c) = [a = c] \text{ and } \phi_2(b, c) = [b = c]$$

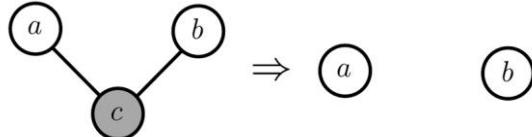
Where  $[\cdot]$  is the Iverson bracket which takes 1 if its arguments is true and 0 otherwise. We obtain the following contradiction:

$$\underbrace{\sum_{a,b,c} \phi_1(a, c) \phi_2(b, c)}_2 \underbrace{\sum_c \phi_1(a, c) \phi_2(b, c)}_{[a=b]} = \underbrace{\sum_{b,c} \phi_1(a, c) \phi_2(b, c)}_1 \underbrace{\sum_{a,c} \phi_1(a, c) \phi_2(b, c)}_1$$

Therefore, in general (for arbitrary choices of the potentials):

$$p(a, b) \neq p(a)p(b)$$

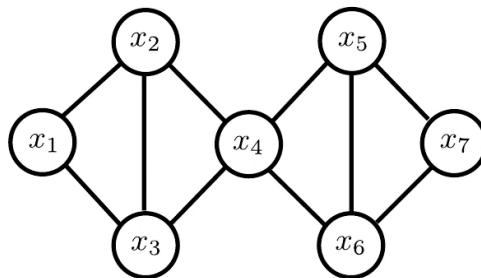
**Conditioning on  $c$  makes  $a$  and  $b$  independent.**



The examples above can be generalized, yielding the **global Markov property**:

**Separation:** A subset  $S$  separates  $A$  from  $B$  if every path from a member of  $A$  to any member of  $B$  passes through  $S$ .

**Global Markov Property:** For disjoint sets of variables  $(A, B, S)$  where  $S$  separates  $A$  from  $B$ , we have  $A \perp B | S$ .



From the global Markov property, we can derive the **local Markov property**:

**Local Markov Property:** When conditioned on its neighbors,  $x$  becomes independent of the remaining variables of the graph:

$$p(x|\chi \setminus \{x\}) = p(x|ne(x))$$

The set of neighbor nodes  $ne(x)$  is called **Markov blanket**. This also holds for sets of variables.

**For example:**

$$p(x_4|x_1, x_2, x_3, x_4, x_5, x_6, x_7) = p(x_4|x_2, x_3, x_5, x_6)$$

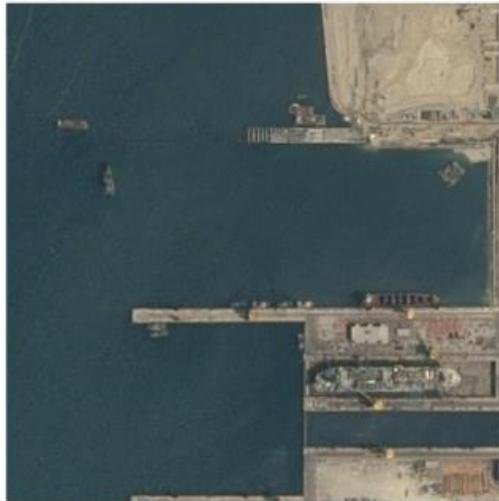
In other words,  $x_4 \perp \{x_1, x_7\} | \{x_2, x_3, x_5, x_6\}$ . Similarly, other independence relationships can be read off the graph.

## Evaluation Metrics

### PIXEL ACCURACY

Pixel accuracy is perhaps the easiest to understand conceptually. It is the **percent of pixels in your image that are classified correctly**.

Here's a scenario: Let's say you ran the following image (Left) through your segmentation model. The image on the right is the ground truth, or annotation (what the model is supposed to segment). In this case, our model is trying to segment ships in a satellite image.



You see that your segmentation accuracy is **95%**. That's awesome! Let's see how your segmentation looks like!



Not exactly what you were hoping for, huh. Is there something wrong with our calculation? Nope. It's exactly right. It's just that one class was 95% of the original image. So if the model classifies all pixels as that class, 95% of pixels are classified accurately while the other 5% are not. As a result, although your accuracy is a whopping 95%, your model is returning a completely useless prediction. This is meant to illustrate that **high pixel accuracy doesn't always imply superior segmentation ability**.

This issue is called **class imbalance**. When our classes are extremely imbalanced, it means that a class or some classes dominate the image, while some other classes make up only a small portion of the image. Unfortunately, class imbalance is prevalent in many real world data sets, so it can't be ignored.

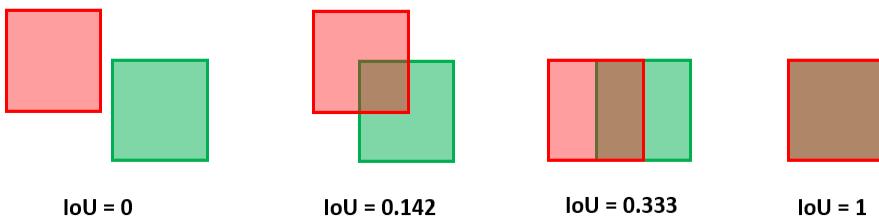
## INTERSECTION OVER UNION

The **Intersection-Over-Union (IoU)**, also known as the **Jaccard Index**, is one of the most commonly used metrics in semantic segmentation. The IoU is a very straightforward metric that's extremely effective.

The **IoU** measures the accuracy of our detections. Given a ground-truth bounding box and a detected bounding box, **we compute the IoU as the ratio of the overlap and union areas**:

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}} = \frac{\text{Intersection}}{\text{Union}}$$

Here are some examples of different IoU values:



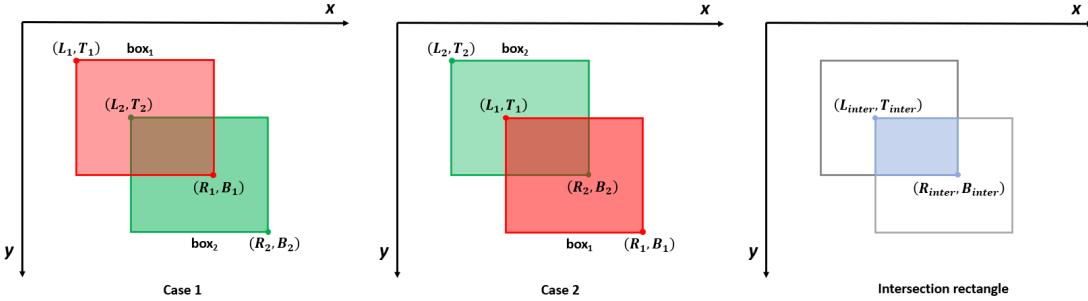
**The IoU can have any value between 0 and 1.** If two boxes do not intersect, the IoU is 0. On the other hand, if they completely overlap, the intersection and the union areas are equal. So, in that case, the IoU is 1.

**Therefore, the higher the IoU, the better the prediction of an object detection system.**

### How to Compute IoU?

We assume a coordinate system with the positive x-axis moving to the right and the positive y-axis moving downward. A bounding box is defined by the left (L), right (R), top (T), and bottom (B).

**Let's start by calculating the coordinates of the intersection rectangle.** Given a pair of bounding boxes, there are two different ways to name the coordinates (cases 1 and 2):



In both cases, the left side of the intersection  $L_{inter}$  is the rightmost left margin of the two bounding boxes. Similarly, the top of the intersection  $T_{inter}$  is the lower top margin of the two boxes. Therefore, the intersection's left-top coordinates are:

$$L_{inter} = \max(L_1, L_2)$$

$$T_{inter} = \max(T_1, T_2)$$

The right side of the intersection  $R_{inter}$  is the leftmost right margin of the two boxes. Similarly, the bottom of the intersection  $B_{inter}$  is the higher bottom margin of the two boxes. Hence, the intersection's bottom-right coordinates are:

$$R_{inter} = \min(R_1, R_2)$$

$$B_{inter} = \min(B_1, B_2)$$

The intersection area  $A_{inter}$  can be easily computed from the obtained coordinates:

$$A_{inter} = (R_{inter} - L_{inter}) \cdot (B_{inter} - T_{inter})$$

**Let's now compute the area of the union.** First we calculate the area of the two boxes:

$$A_1 = (R_1 - L_1) \cdot (B_1 - T_1)$$

$$A_2 = (R_2 - L_2) \cdot (B_2 - T_2)$$

The union area  $A_{union}$  is computed as:

$$A_{union} = A_1 + A_2 - A_{inter}$$

it is worth noting that intersection area is included in both  $A_1$  and  $A_2$ , hence  $A_{inter}$  is subtracted to  $(A_1 + A_2)$  in order to count the intersection area only once.

**Finally, we can calculate the IoU:**

$$IoU = \frac{A_{inter}}{A_{union}}$$

Example

Let's consider the two boxes with coordinates:

$$box_1 = [L_1, T_1, R_1, B_1] = [0, 0, 10, 10]$$

$$box_2 = [L_2, T_2, R_2, B_2] = [5, 5, 15, 15]$$

The coordinates of the intersection rectangle are:

$$[L_{inter}, T_{inter}, R_{inter}, B_{inter}] = [5, 5, 10, 10]$$

So, the intersection area is:

$$A_{inter} = (10 - 5) \cdot (10 - 5) = 5 \cdot 5 = 25$$

Each box has an area of 100, so the union area is:

$$A_{union} = 100 + 100 - 25 = 175$$

Finally, the IoU is:

$$IoU = \frac{25}{175} = 0.142$$

Computational Complexity

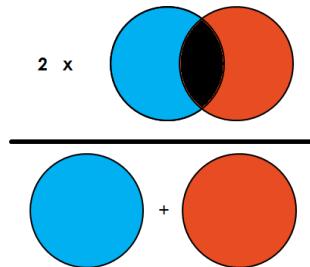
The algorithm always requires 8 input values. Hence, the spatial complexity and time complexity are both constant, i.e.,  $o(1)$ .

The segmentation challenge is evaluated using the **mean Intersection over Union** (mIoU) metric.

$$mIoU = \frac{TP}{TP + FP + FN}$$

DICE COEFFICIENT (F1 SCORE)

Simply put, the Dice Coefficient is 2 \* the Area of Overlap divided by the total number of pixels in both images.

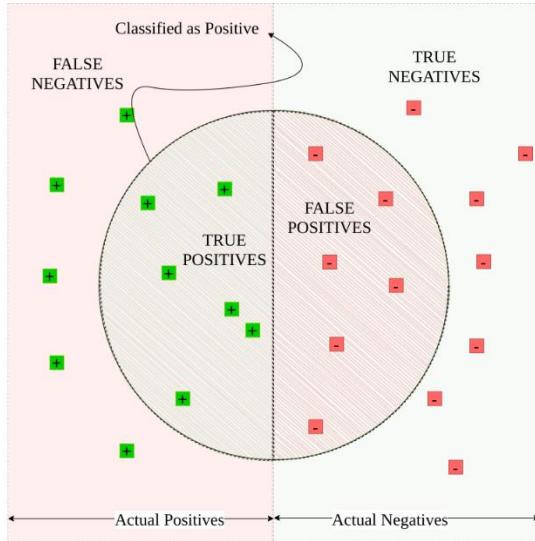


## BINARY CLASSIFICATION MEASURES

Suppose we have a simple binary classification case as shown in the figure below. The actual positive and negative samples are distributed on rectangular surfaces.

However, our classifier marks the samples inside the circle as positive and the rest as negative.

How did our classifier do?



Many metrics are proposed and being used to evaluate the performance of a classifier.

**Depending on the problem and goal, we must select the relevant metric to observe and report.**

In this section, we'll try to understand what do some popular classification measures represent and which one to use for different problems.

### *Accuracy*

The most simple and straightforward classification metric is accuracy. **Accuracy measures the fraction of correctly classified observations.** The formula is:

$$\text{Accuracy} = \frac{TP + TN}{\text{Number of Samples}} = \frac{TP + TN}{TP + TN + FP + FN}$$

Thinking in terms of the confusion matrix, we sometimes say “the diagonal over all samples”.

**Simply put, it measures the absence of error rate.** An accuracy of 90% means that out of 100 observations, 90 samples are classified correctly.

Although 90% accuracy sounds very promising at first, using the accuracy measure in imbalanced datasets might be misleading.

If we talk about spam e-mail example. Consider out of every 100 e-mails received, 90 of them are spam. In this case, labeling each e-mail as spam would lead to 90% accuracy with an empty inbox. Think about all the important e-mails we might be missing.

Similarly, in the case of fraud detection, only a tiny fraction of transactions are fraudulent. If our classifier were to mark every single case as not fraudulent, we'd still have close to 100% accuracy.

### *Precision and Recall*

An alternative measure to accuracy is precision. **Precision is the fraction of instances marked as positive that are actually positive. In other words, precision measures “how useful are the results of our classifier”.** The mathematical notation is:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Otherwise speaking, a precision of 90% would mean when our classifier marks an e-mail as spam, it is indeed spam 90 out of 100 times.

Another way to look at the TPs is by using recall. **Recall is the fraction of true positive instances that are marked to be positive. It measures “how complete the results are” — that is, which percentage of true positives are predicted as positive.** The representation is:

$$\text{Recall} = \frac{TP}{TP + FN}$$

That is to say, a recall of 90% would mean that the classifier correctly labels 90% of all the spam e-mails, hence 10% are marked as not spam.

Classical classifiers have a threshold value, for which they mark the lower results as negative and higher results as positive. Changing the defined threshold value for a binary classifier leads to changes in the predicted labels. So, reducing the rate of one error type leads to an increase in the other one. As a result, there is a trade-off between precision and recall.

All in all, both [precision and recall](#) measure the absence of errors in special terms. **Perfect precision is equivalent to no FPs (no Type I errors), while on the other hand, perfect recall means there are no FNs (no Type II errors).** In some cases, we need to select the relevant error type to consider, depending on the problem goal.

The **mean Average Precision (mAP)** is the mean of the Average Precisions computed over all the classes of the challenge:

- Let  $Q$  be the number of classes, and  $\text{AvgP}(q)$  is the average precision of the  $q^{th}$  category:

$$mAP = \frac{\sum_{q=1}^Q \text{AvgP}(q)}{Q}$$

The mAP metric avoids to have extreme specialization in few classes and thus weak performances in others.

### F-1 Score

Usually, having a balanced recall and precision is more important than having one type of error really low. That's why generally we take both precision and recall into consideration. Several classification measures are developed to identify this problem.

For example, the **F-1 score** is the harmonic mean of precision and recall. It gives equal importance to Type I and Type II errors. The calculation is:

$$F - 1 \text{ Score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

When the dataset labels are evenly distributed, accuracy gives meaningful results. But if the dataset is imbalanced, like the spam e-mail example, we should prefer the F-1 score.

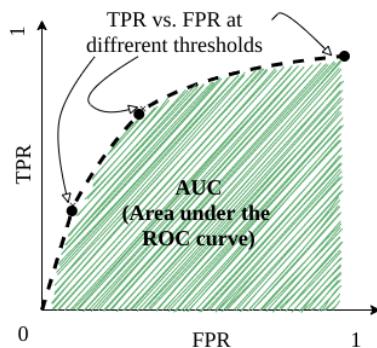
Another point to keep in mind is; accuracy gives more importance to TPs and TNs. On the other hand, the F-1 score considers FN and FPs.

### ROC Curve and AUC

A well-known method to visualize the classification performance is a **ROC curve (receiver operating characteristic curve)**. The plot shows the classifier's success for different threshold values.

In order to plot the ROC curve, we need to calculate the True Positive Rate (*TPR*) and the False Positive Rate (*FPR*), where:

$$\begin{aligned} TPR &= \frac{TP}{TP + FN} \\ FPR &= \frac{FP}{FP + TN} \end{aligned}$$



To better understand what this graph stands for, first imagine setting the classifier threshold to  $-\infty$ . This will lead to labeling every observation as positive. Thus, we can conclude that lowering the threshold marks more items as positive. Having more observations classified as positive results in having more TPs and FPs.

Conversely, setting the threshold value to  $\infty$  causes every observation to be marked as negative. Then, the algorithm will label more observations as negative. So, increasing the threshold value will cause both TP and FP numbers to decrease, leading to lower TPR and FPR.

For each different threshold value, we get different TP and FP rates. Plotting the result gives us the ROC curve. But how does the ROC curve help us find out the success of a classifier?

In this case, AUC helps us. It stands for “area under the curve”. **Simply put, AUC is the area under the ROC curve. In case we have a better classification for each threshold value, the area grows. A perfect classification leads to an AUC of 1.0.**

On the contrary, worse classifier performance reduces the area. AUC is often used as a classifier comparison metric for machine learning problems.

## MULTICLASS CLASSIFICATION MEASURES

**When there are more than two labels available for a classification problem, we call it multiclass classification.** Measuring the performance of a multiclass classifier is very similar to the binary case.

		Actual Classes			
		a	b	c	d
Predicted Classes	a	50	3	0	0
	b	26	8	0	1
	c	20	2	4	0
	d	12	0	0	1

Suppose a certain classifier generates the confusion matrix presented above. There are 127 samples in total. Now let's see how well the classifier performed.

Recall that accuracy is the percentage of correctly classified samples, which reside on the diagonal. **To calculate accuracy, we need to just divide the number of correctly classified samples by the total number of samples.**

$$\text{Accuracy} = \frac{\text{Number of correctly classified}}{\text{Number of samples}} = \frac{50 + 8 + 4 + 1}{127} = 49.6\%$$

So, the classifier correctly classified almost half of the samples. Given we have five labels, it is much better than the random case.

Calculating the precision and recall is a little trickier than the binary class case. **We can't talk about the overall precision or recall of a classifier for multiclass classification. Instead, we calculate precision and recall for each class separately.**

Class by class, we can categorize each element into one of TP, TN, FP, or FN. Hence, we can rewrite the precision formula as:

$$\begin{aligned} \text{Precision}(class = a) &= \frac{\text{TP for class } a}{\text{Number of classified as class } a} = \\ &= \frac{\text{TP}(class = a)}{\text{TP}(class = a) + \text{FP}(class = a)} = \frac{50}{50 + 3} = 94.3\% \end{aligned}$$

Similarly, the recall formula can be rewritten as:

$$\begin{aligned} \text{Recall}(class = a) &= \frac{\text{TP for class } a}{\text{Number of actual class } a} = \\ &= \frac{\text{TP}(class = a)}{\text{TP}(class = a) + \text{FN}(class = a)} = \frac{50}{50 + 26 + 20 + 12} = 46.3\% \end{aligned}$$

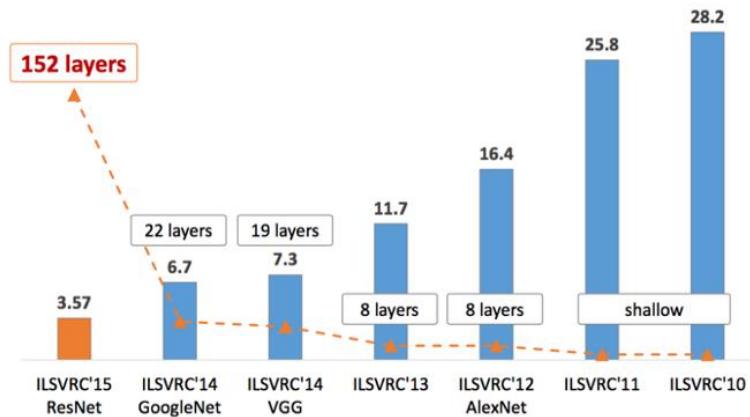
Correspondingly, we can calculate precision and recall values for other classes.

$$\begin{array}{ll} \text{Precision}(class = b) = \frac{8}{35}, & \text{Recall}(class = b) = \frac{8}{13} \\ \text{Precision}(class = c) = \frac{4}{26}, & \text{Recall}(class = c) = \frac{4}{4} \\ \text{Precision}(class = d) = \frac{1}{13}, & \text{Recall}(class = d) = \frac{1}{2} \end{array}$$

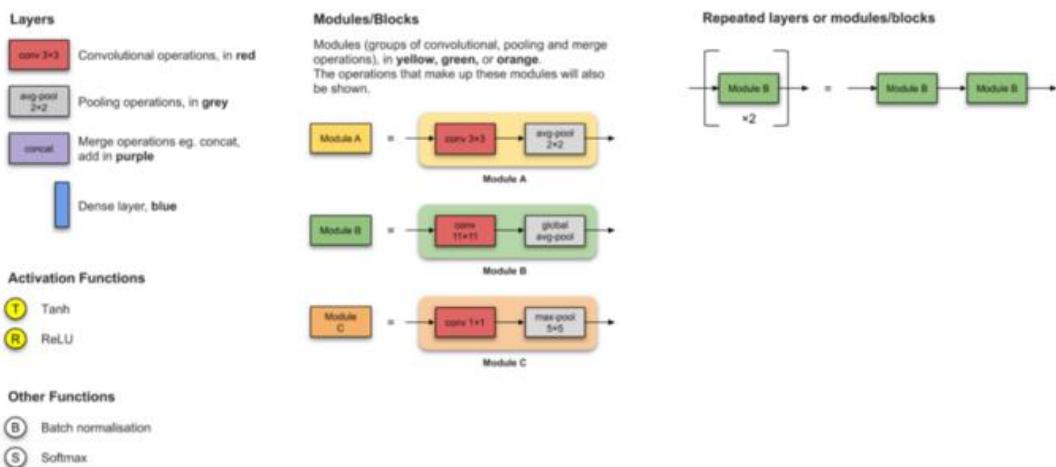
## TOOLS

In both semantic and instance segmentations, deep-learning methods have achieved the best performance.

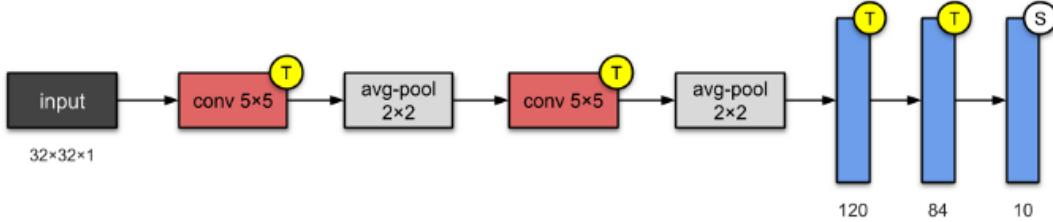
The **ImageNet** project is a large visual database designed for use in visual object recognition software research. The ImageNet project runs an annual software contest, the **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)**, where software programs compete to correctly classify and detect objects and scenes. Here I will talk about CNN architectures of ILSVRC top competitors.



Legend:



## LeNet-5 (1998)

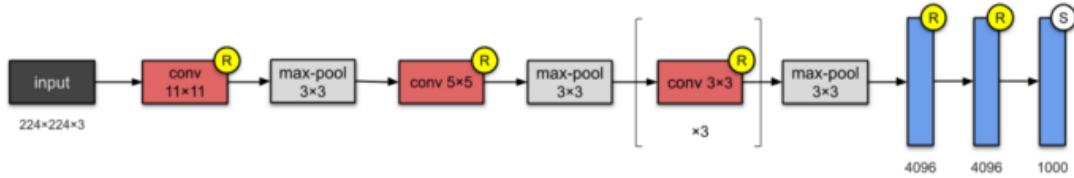


LeNet-5 is one of the simplest architectures. It has 2 convolutional and 3 fully-connected layers (hence “5” — it is very common for the names of neural networks to be derived from the number of *convolutional* and *fully connected* layers that they have). The average-pooling layer as we know it now was called a *sub-sampling layer* and it had trainable weights (which isn’t the current practice of designing CNNs nowadays). This architecture has about **60,000 parameters**.

### WHAT'S NOVEL?

This architecture has become the standard ‘template’: stacking convolutions with activation function, and pooling layers, and ending the network with one or more fully-connected layers.

## AlexNet (2012)



With **60M parameters**, AlexNet has 8 layers — 5 convolutional and 3 fully connected. AlexNet just stacked a few more layers onto LeNet-5. At the point of publication, the authors pointed out that their architecture was “one of the largest convolutional neural networks to date on the subsets of ImageNet.”

### WHAT'S NOVEL?

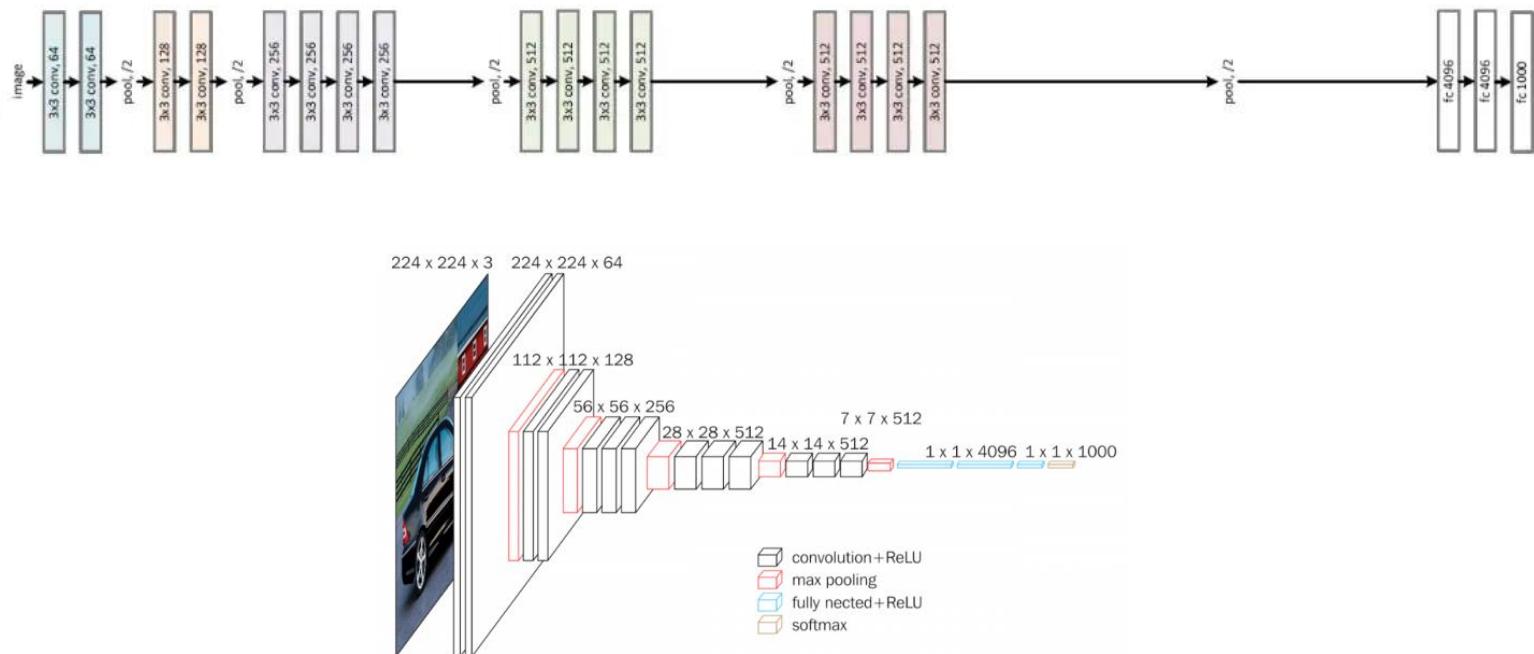
0. They were the first to implement Rectified Linear Units (ReLUs) as activation functions.
1. Dropout.

## VGG Network

**VGG16** is a **convolutional neural network** that was used in the ImageNet competition in 2014.

One interesting part of this network is that it **used only 3x3 filters** even in the first layers. If we compare it to the previous ones, [AlexNet](#), the winner in 2012, used 11x11 and 5x5 filters at the first two layers. And [ZFnet](#), the winner in 2013, used 7x7 filters. Instead of using a relatively large size of a filter in the first convolution layer, the size of the filters VGG network used is 3x3 only.

So what's the intention for trying this? The answer is to reduce the number of parameters. Let's do some calculation here. How many parameters does a single 7x7 convolution layer with the channel size C require? It's  $49C^2$ . Now, what's the number with 3 layers of 3x3 convolution layers with the same channel size?  $(9C^2) \times 3 = 27C^2$ . See how many parameters decreases. Reducing the number of weights can handle overfitting by reducing the complexity of the network. Moreover, as we can have an activation layer for each convolution layer, this encourages the model's learning as well.



VGG16 - Structural Details													
#	Input Image		output	Layer	Stride	Kernel	in	out	Param				
1	224	224	3	224	224	64	conv3-64	1	3	3	3	64	1792
2	224	224	64	224	224	64	conv3064	1	3	3	64	64	36928
	224	224	64	112	112	64	maxpool	2	2	2	64	64	0
3	112	112	64	112	112	128	conv3-128	1	3	3	64	128	73856
4	112	112	128	112	112	128	conv3-128	1	3	3	128	128	147584
	112	112	128	56	56	128	maxpool	2	2	2	128	128	65664
5	56	56	128	56	56	256	conv3-256	1	3	3	128	256	295168
6	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
7	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
	56	56	256	28	28	256	maxpool	2	2	2	256	256	0
8	28	28	256	28	28	512	conv3-512	1	3	3	256	512	1180160
9	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
10	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
	28	28	512	14	14	512	maxpool	2	2	2	512	512	0
11	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
12	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
13	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
	14	14	512	7	7	512	maxpool	2	2	2	512	512	0
14	1	1	25088	1	1	4096	fc	1	1	25088	4096	102764544	
15	1	1	4096	1	1	4096	fc	1	1	4096	4096	16781312	
16	1	1	4096	1	1	1000	fc	1	1	4096	1000	4097000	
Total								138,423,208					

Along with the VGG networks or AlexNet, most of the Convolutional networks followed the basic structure: a series of convolutional layers, a pooling layer and an activation layer with some of the normalization. But going deeper and deeper, people came across some serious problems and started to redesign this approach.

### WHAT'S NOVEL?

As mentioned in their abstract, the contribution from this paper is the designing of *deeper* networks (roughly twice as deep as AlexNet). This was done by stacking uniform convolutions.

## ResNet

Intuitively we'd like to expect "deeper is better." However, researchers found that it wasn't the case. **As a network gets extremely deeper and deeper, the performance gets worse.** One of the reasons is a **vanishing/ exploding gradient problem**.

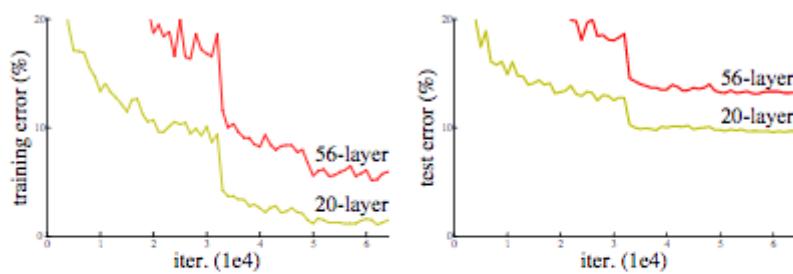


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

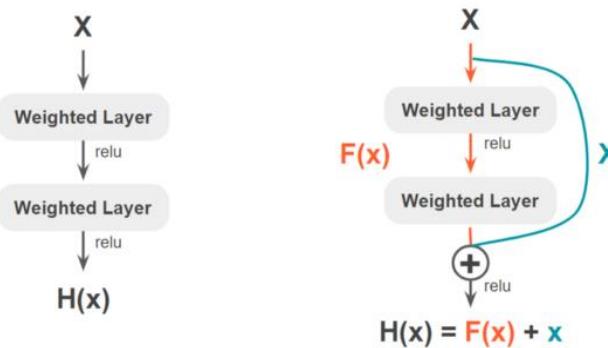
Although we already have several ways to address this issue such as gradient clipping, there is a degradation problem that isn't due to overfitting. **The birth of ResNet started from here.** Why is that?

Let's assume that we have a good network with desirable accuracy. Now we're going to copy this network and add just 1 layer but having no additional function. This is called an **identity mapping**, which means getting the output exactly the same with the input. Compared to other complex layers with many parameters, this will be a piece of cake to our “smart” model. So there should be no harm to the performance of the network, right?



Then what will be the outcome if we attach more and more identity layers? As identity layers give no impact to the output, there should be no degradation in the performance. It seems a reasonable guess, but the outcome of this experiment was different from our expectations. The performance of this model was worse than the shallower model without identity layers.

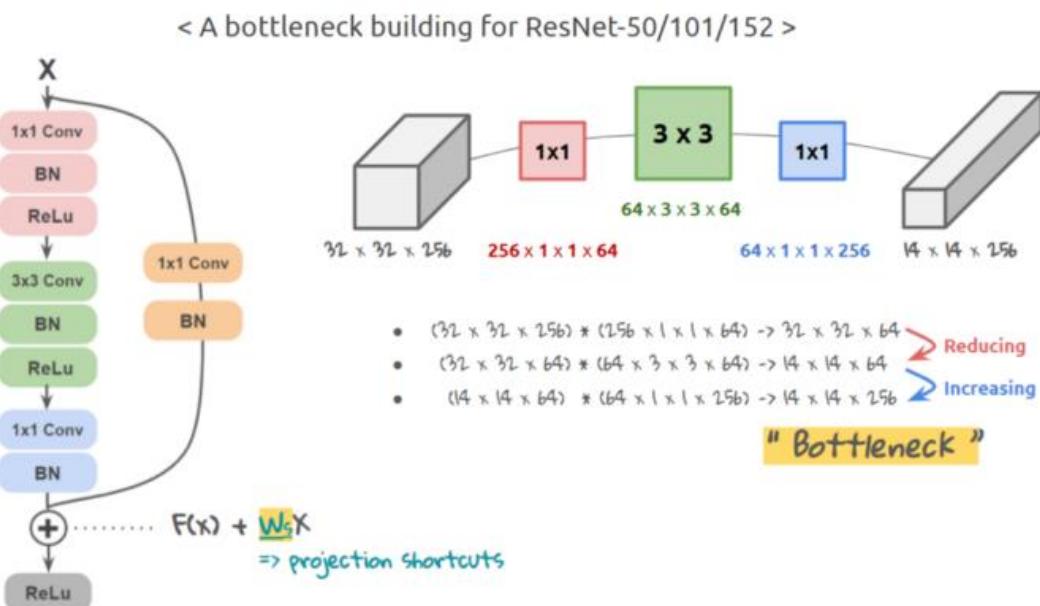
This outcome indicates “deeper is better” is not applicable to neural networks even with the identity mapping. Just attaching identity layers directly to a network makes it hard to train. Okay, then **what about putting identity layers as an additional layer?**



### SKIP/SHORTCUT CONNECTION (IDENTITY LAYER)

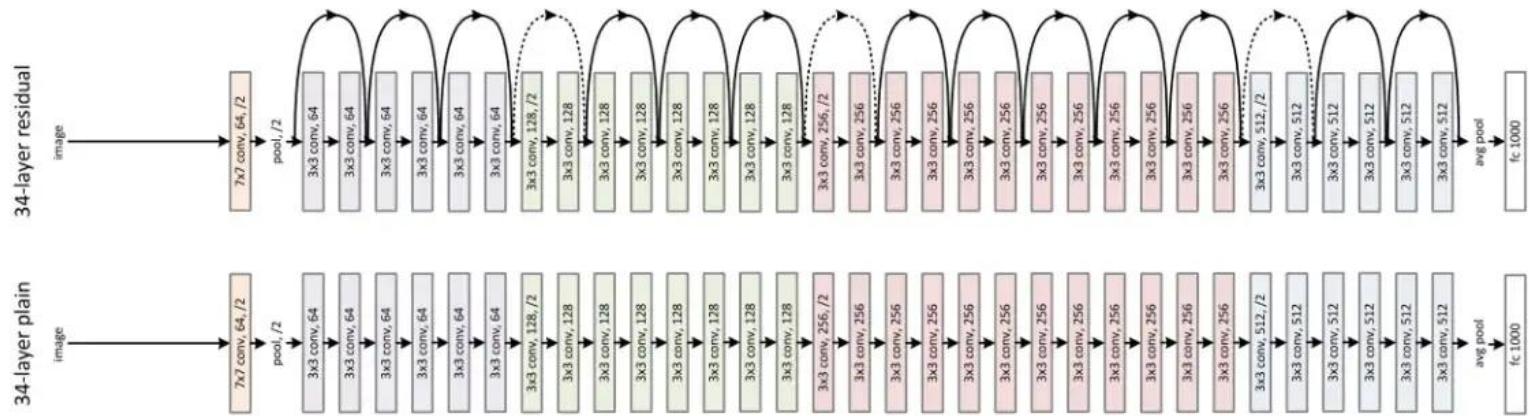
Now, this is where ResNet comes into play. Let's say  $H(x)$  as the outcome of the network. And if we put an additional path, the original function becomes  $H(x) = F(x) + x$  where  $F(x)$  is the function of the **main branch** as shown above. The identity layer is also called a **shortcut connection** or a **skip connection**. This brings the concept of “**residual learning**” to the network. Now the main branch is trained to approximate  $H(x) - x$  and make  $F(x)$  go closer to zero. This means giving the layers a certain direction for learning, which makes the training easier. Moreover, the network can skip some of the layers through the identity layer. This allows us to have deeper layers but still with high performance.

You may ask how it's possible to add  $F(x)$  and  $x$  when the two layers have different sizes. We can handle this problem by the zero padding or a linear projection at the shortcut connections. So there are two kinds of shortcut connections in ResNet, one without weights and the other with weights.



The picture above shows the convolutional block of ResNet 50-layers. It has a main branch with 3 convolution layers and a shortcut connection with a 1x1 convolution. Here we may need to ask, **why 1x1-3x3-1x1 convolution?**

To answer this question, it is necessary to understand how the dimension of data changes according to filter size. Please have a look at the picture on the right. If we say the shape of the input is (32,32,256), the dimension decreases when it passes the 3x3 filter compared to the previous step. And when it passes the second 1x1 filter, it increases again. This is making a bottleneck. By forcing the data to fit into a smaller dimension, we can get a more efficient feature representation.

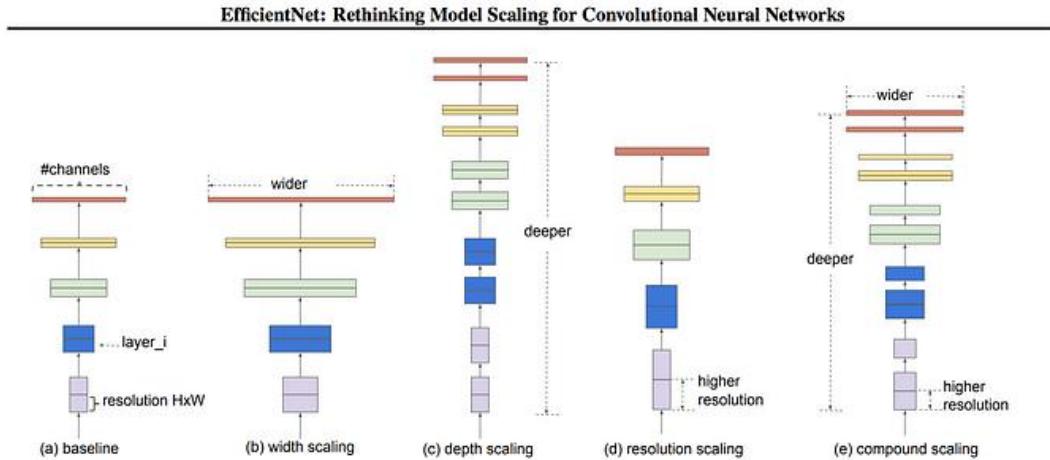


Optimizing the deep neural network easily with much fewer filters, much less computation, and higher accuracy. This simple yet novel reformulation brought such a beautiful achievement is how ResNet blew people's minds in 2015. With this fundamental intuition on ResNet, I'd like you to explore the network from the [original paper](#). And [here](#) is the second part of the research on experimenting the variants of the residual block.

## EfficientNet

### SCALING IN CNN

There are three scaling dimensions of a CNN: **depth**, **width**, and **resolution**. **Depth** simply means how deep the networks is which is equivalent to the number of layers in it. **Width** simply means how wide the network is. One measure of width, for example, is the number of channels in a Conv layer whereas **Resolution** is simply the image resolution that is being passed to a CNN. The figure below (from the paper itself) will give you a clear idea of what scaling means across different dimensions. We will discuss these in detail as well.



### Depth Scaling

Scaling a network by depth is the most common way of scaling. Depth can be scaled up as well as scaled down by adding/removing layers respectively. For example, ResNets can be scaled up from ResNet-50 to ResNet-200 as well as they can be scaled down from ResNet-50 to ResNet-18. But why depth scaling? The intuition is that a **deeper network can capture richer and more complex features, and generalizes well on new tasks**.

Theoretically, with more layers, the network performance should improve but practically it doesn't follow. **Vanishing gradients is one of the most common problems that arises as we go deep**. Even if you avoid the gradients to vanish, as well as use some techniques to make the training smooth, adding more layers doesn't always help. For example, ResNet-1000 has similar accuracy as ResNet-101.

### Width Scaling

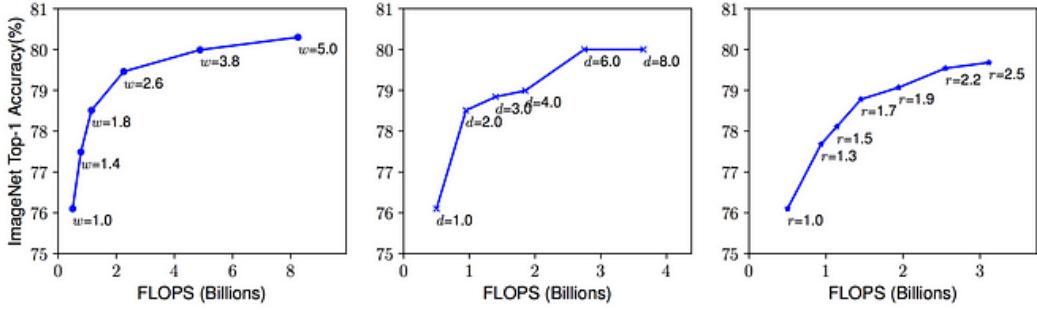
This is commonly used when we want to keep our model small. **Wider networks tend to be able to capture more fine-grained features**. Also, **smaller models are easier to train**.

The problem is that even though you can make your network extremely wide, with shallow models (less deep but wider) **accuracy saturates quickly with larger width**.

### Resolution

Intuitively, we can say that **in a high-resolution image, the features are more fine-grained and hence high-res images should work better**. This is also one of the reasons that in complex tasks, like Object detection, we use image resolutions like 300x300, or 512x512, or 600x600. **But this doesn't scale linearly. The accuracy gain diminishes very quickly**. For example, increasing resolution from 500x500 to 560x560 doesn't yield significant improvements.

The above three points lead to our **first observation: Scaling up any dimension of network (width, depth or resolution) improves accuracy, but the accuracy gain diminishes for bigger models**.

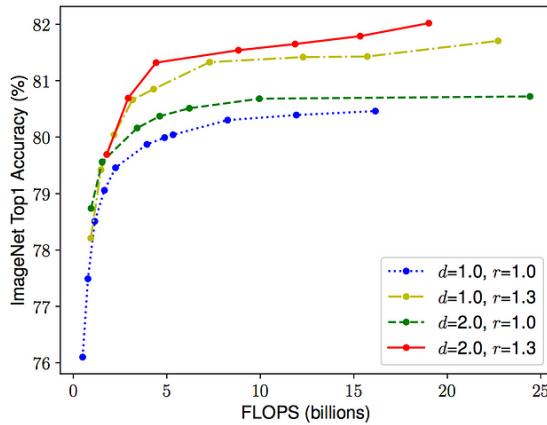


## COMBINED SCALING

We can combine the scalings for different dimensions but there are some points that the authors have made:

- Though it is possible to scale two or three dimensions arbitrarily, arbitrary scaling is a tedious task.
- Most of the times, manual scaling results in sub-optimal accuracy and efficiency.

Intuition says that as the resolution of the images is increased, depth and width of the network should be increased as well. As the depth is increased, larger receptive fields can capture similar features that include more pixels in an image. Also, as the width is increased, more fine-grained features will be captured. To validate this intuition, the authors ran a number of experiments with different scaling values for each dimension. For example, as shown in the figure below from the paper, with deeper and higher resolution, width scaling achieves much better accuracy under the same FLOPS cost.



These results lead to our **second** observation: **It is critical to balance all dimensions of a network (width, depth, and resolution) during CNNs scaling for getting improved accuracy and efficiency.**

### *Proposed Compound Scaling*

The authors proposed a simple yet very effective scaling technique which uses a compound coefficient  $\phi$  to uniformly scale network width, depth, and resolution in a principled way:

$$\begin{aligned}
 \text{depth: } d &= \alpha^\phi \\
 \text{width: } w &= \beta^\phi \\
 \text{resolution: } r &= \gamma^\phi \\
 \text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 &\approx 2 \\
 \alpha \geq 1, \beta \geq 1, \gamma \geq 1
 \end{aligned}$$

$\phi$  is a user-specified coefficient that controls how many resources are available whereas  $\alpha$ ,  $\beta$ , and  $\gamma$  specify how to assign these resources to network depth, width, and resolution respectively.

In a CNN, Conv layers are the most compute expensive part of the network. Also, FLOPS of a regular convolution op is almost proportional to  $d \cdot w^2 \cdot r^2$ , i.e. doubling the depth will double the FLOPS while doubling width or resolution increases FLOPS almost by four times. Hence, in order to make sure that the total FLOPS don't exceed  $2^\phi$ , the constraint applied is that  $(\alpha * \beta^2 * \gamma^2) \approx 2$ .

## EFFICIENTNET ARCHITECTURE

Scaling doesn't change the layer operations, hence it is better to first have a good baseline network and then scale it along different dimensions using the proposed compound scaling. The authors obtained their base network by doing a Neural Architecture Search (NAS) that optimizes for both accuracy and FLOPS. The architecture is similar to M-NASNet as it has been found using the similar search space. The network layers/blocks are as shown below:

Stage $i$	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	$224 \times 224$	32	1
2	MBConv1, k3x3	$112 \times 112$	16	1
3	MBConv6, k3x3	$112 \times 112$	24	2
4	MBConv6, k5x5	$56 \times 56$	40	2
5	MBConv6, k3x3	$28 \times 28$	80	3
6	MBConv6, k5x5	$28 \times 28$	112	3
7	MBConv6, k5x5	$14 \times 14$	192	4
8	MBConv6, k3x3	$7 \times 7$	320	1
9	Conv1x1 & Pooling & FC	$7 \times 7$	1280	1

The MBConv block is nothing fancy but an Inverted Residual Block (used in MobileNetV2) with a Squeeze and Excite block injected sometimes.

Now we have the base network, we can search for optimal values for our scaling parameters. If you revisit the equation, you will quickly realize that we have a total of four parameters to search

for:  $\alpha$ ,  $\beta$ ,  $\gamma$ , and  $\varphi$ . In order to make the search space smaller and making the search operation less costly, the search for these parameters can be completed in two steps.

1. Fix  $\varphi = 1$ , assuming that twice more resources are available, and do a small grid search for  $\alpha$ ,  $\beta$ , and  $\gamma$ . For baseline network B0, it turned out the optimal values are  $\alpha = 1.2$ ,  $\beta = 1.1$ , and  $\gamma = 1.15$  such that  $\alpha * \beta^2 * \gamma^2 \approx 2$
2. Now fix  $\alpha$ ,  $\beta$ , and  $\gamma$  as constants (with values found in above step) and experiment with different values of  $\varphi$ . The different values of  $\varphi$  produce EfficientNets B1-B7.

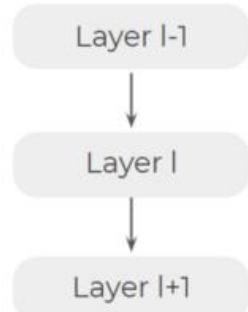
## GoogLeNet

Going deeper with high performance is good. We need deep neural networks to implement complex and challenging tasks. But there is another factor to consider when we evaluate the networks. Think about when we have to embed a model in a mobile application. It'll be not desirable to have a "too heavy" model for sure. But more than that, there is a computational cost problem that is quite critical in real practice. So now the question becomes, how **can we go deeper with higher efficiency?**

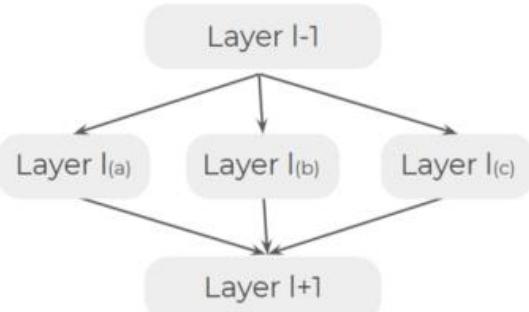


One place where the researchers had found **inefficient computation occurred was a fully connected architecture**. In a convolutional approach, a fully connected architecture indicates that a layer  $l$  feeds from a layer  $l - 1$  and is connected to the next layer  $l + 1$  as shown below on the left. The researchers found any uniform increase in the number of the filters doubles the computation. So they suggest moving from densely connected architectures to sparsely connected architectures.

*"Fully connected"*



*"Sparsely connected"*

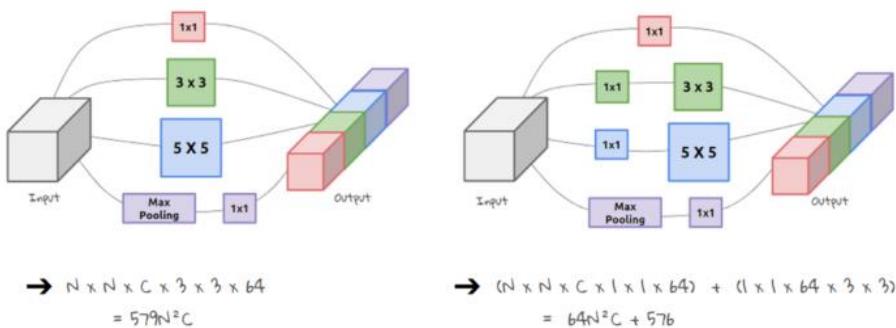


Instead of implementing one **type** of convolutions, the sparse architecture does multiple convolutions with multiple filter sizes as shown above. And the dimension of output from these layers can be matched up with ‘SAME’ padding.

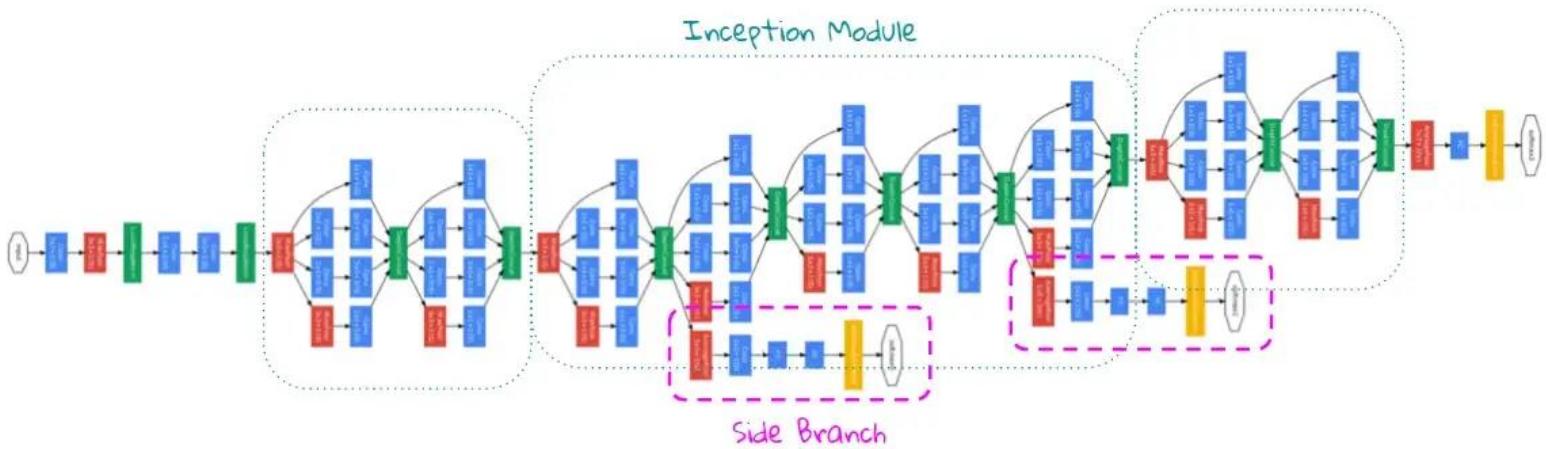
There is an important meaning in having several filters with different sizes here. During the training, the scale of features a network detects are small and local at first, and it gets bigger as it goes deeper. For example, let’s say the model detects a cat. The model sees a small scale of features such as the fur pattern of the cat, and then it moves to a bigger scale such as the shape of its ear, and then the number of legs.



So even with the pictures where the cats are at different scales like above, the model still processes the same scale of feature mapping. This is inefficient and it hurts the performance. Therefore having multiple filters indicates now it knows how to “zoom in and out” according to the input image. It **enables the model to detect the image in different scales at a time**. And by doing so, we can also **handle off the work of choosing parameters as well as reducing the inefficient computation**. This is the first kick of **Inception network**.



So the **Inception Module** looked like the one on the left of the picture above. But there was a problem. **What about the computation cost?** Sparsely connected architectures still require the same or more amount of computation anyways! The solution of the researchers to handle this problem was implementing 1x1 convolutions as shown above on the right. **The purpose of the 1x1 convolutions here is dimension reduction** just as what we discussed with ResNet. See the computation times are reduced given the  $(N, N, C)$  shape of the input image. If the size of the image is  $32 \times 32$  with the 129 channels, we can reduce the number of parameters 9 times less!



The final architecture of GoogleNet is shown above. The modules are concatenated, forming 22 layers in total. This is quite a deep network and it also couldn't avoid the notorious vanishing gradient problem. So the way the researchers took to handle this issue was making side branches.

What do we mean by “**gradient vanish**?” It indicates that the value of the gradient gets smaller and smaller as it goes deeper, which means **the gradient has little information to learn**. So we can say that the gradient from the early stages carries more information. In other words, **we can have additional information by adding helper classifiers in the intermediate layers**. Moreover, as we discussed above, the features detected are different along with the layers. Therefore the **“intermediate products”** are also beneficial for classifying the labels. We add the loss from these branches to the total loss of the network with weighted.

## MobileNet

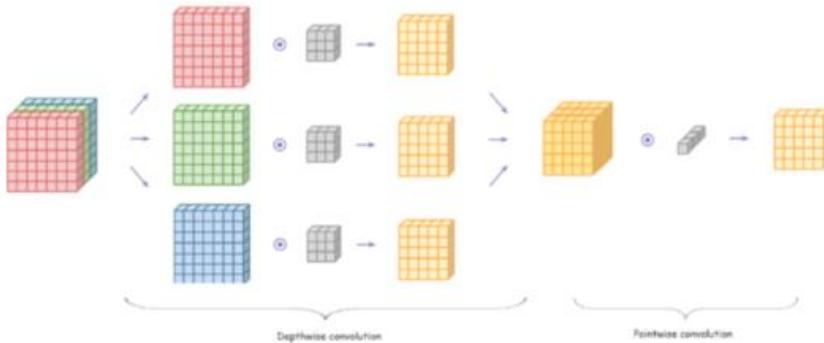
Inspired by the networks we discussed above, “going deeper and more complicated” was the high trend in CNN research. Although those networks brought higher accuracy, **they were not efficient or ‘light’ enough for a real-world application**. To be carried out in a computationally limited platform with a timely manner, they need to be smaller and faster.

One solution the researchers proposed was forming a **factorized convolutions**. Factorizing convolutions indicates factorizing a standard convolution into a depthwise convolution and a pointwise convolution. This is called **depthwise separable convolution**.

- Standard Convolution



- Depthwise Separable Convolution



In depthwise separable convolution, we first split each channel of the input and the filter, and convolve the input with the corresponding filter, or in a depthwise manner. And then concatenate the output together. This is **depthwise convolution**. After that, we do **pointwise convolution**, which is the same as 1x1 convolution. Now see the final outcome. Isn't it the same with that of a standard 3x3 convolution? If so, what's the point for doing this?

### < Computational Cost Comparison >

① Standard Convolution

$$: N \cdot N \cdot C_1 \cdot C_2 \cdot F \cdot F$$

② Depthwise Separable Convolution:

$$: (N \cdot N \cdot F \cdot F) \cdot C_1 + C_1 \cdot F \cdot F \cdot C_2$$

$$\rightarrow \frac{(N \cdot N \cdot F \cdot F) \cdot C_1 + C_1 \cdot F \cdot F \cdot C_2}{N \cdot N \cdot C_1 \cdot C_2 \cdot F \cdot F} = \frac{1}{C_2} + \frac{1}{N^2}$$

The answer is computation. Let's compare the computational cost of the two. When the shape of input is  $(N, N, C_1)$  and the output channel is  $C_2$  with the filter size  $F$ , the total computation of the two should be as shown on the left. So as you can see, the computational cost is reduced with depthwise separable convolution. According to the [original paper](#), MobileNet requires 8 to 9 times less computation with only a small reduction in accuracy compared to the same architecture but with standard convolutions.

Depthwise separable convolution is also used to **Xception**, and its paper was released earlier than that of MobileNet. But I found MobileNet paper gives a more straightforward explanation on the concept of depthwise separable convolution.

## SEPERABLE CONVOLUTIONS

There are two main types of separable convolutions: spatial separable convolutions, and depthwise separable convolutions.

### *Spatial Separable Convolutions*

Conceptually, this is the easier one out of the two, and illustrates the idea of **separating one convolution into two**. Unfortunately, spatial separable convolutions have some significant limitations, meaning that it is not heavily used in deep learning.

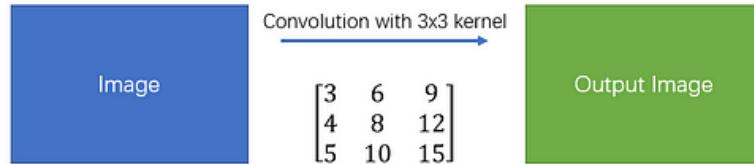
The **spatial separable convolution** is so named because it deals primarily with the **spatial dimensions** of an image and kernel: the width and the height. (The other dimension, the “depth” dimension, is the number of channels of each image).

A spatial separable convolution simply divides a kernel into two, smaller kernels. The most common case would be to divide a 3x3 kernel into a 3x1 and 1x3 kernel, like so:

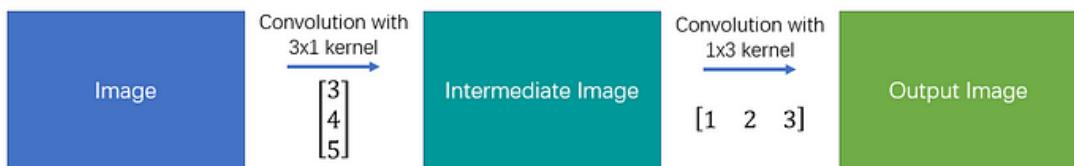
$$\begin{bmatrix} 3 & 6 & 9 \\ 4 & 8 & 12 \\ 5 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \times [1 \ 2 \ 3]$$

Now, instead of doing one convolution with 9 multiplications, we do two convolutions with 3 multiplications each (6 in total) to achieve the same effect. With less multiplications, computational complexity goes down, and the network is able to run faster.

## Simple Convolution



## Spatial Separable Convolution



One of the most famous convolutions that can be separated spatially is the Sobel kernel, used to detect edges:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times [-1 \ 0 \ 1]$$

The main issue with the spatial separable convolution is that not all kernels can be “separated” into two, smaller kernels. This becomes particularly bothersome during training, since of all the possible kernels the network could have adopted, it can only end up using one of the tiny portion that can be separated into two smaller kernels.

### *Depthwise Separable Convolutions*

Unlike spatial separable convolutions, depthwise separable convolutions work with kernels that cannot be “factored” into two smaller kernels. Hence, it is more commonly used.

The depthwise separable convolution is so named because it deals not just with the spatial dimensions, but with the depth dimension — the number of channels — as well. An input image may have 3 channels: RGB. After a few convolutions, an image may have multiple channels. You can image each channel as a particular interpretation of that image; in for example, the “red” channel interprets the “redness” of each pixel, the “blue” channel interprets the “blueness” of each pixel, and the “green” channel interprets the “greenness” of each pixel. An image with 64 channels has 64 different interpretations of that image.

Similar to the spatial separable convolution, a depthwise separable convolution splits a kernel into 2 separate kernels that do two convolutions: the depthwise convolution and the pointwise convolution. But first of all, let’s see how a normal convolution works.

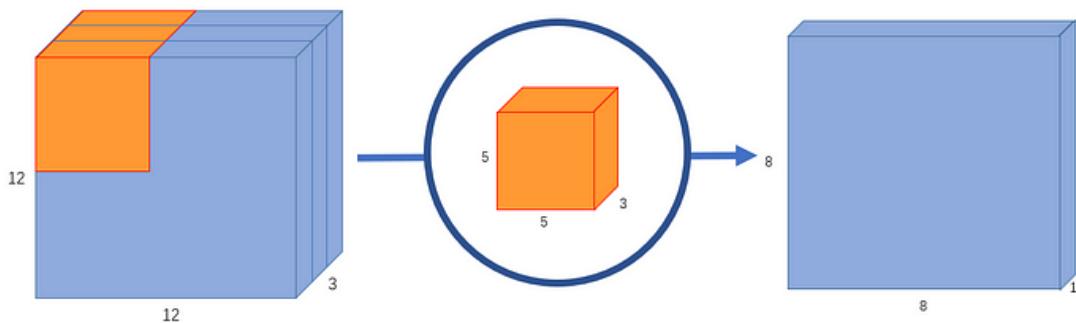
#### Normal Convolution

Let us assume that we have an input image of 12x12x3 pixels, an RGB image of size 12x12.

Let’s do a 5x5 convolution on the image with no padding and a stride of 1. If we only consider the width and height of the image, the convolution process is kind of like this: 12x12 — (5x5) — >8x8. The 5x5 kernel undergoes scalar multiplication with every 25 pixels, giving out 1 number every time. We end up with a 8x8 pixel image, since there is no padding ( $12 - 5 + 1 = 8$ ).

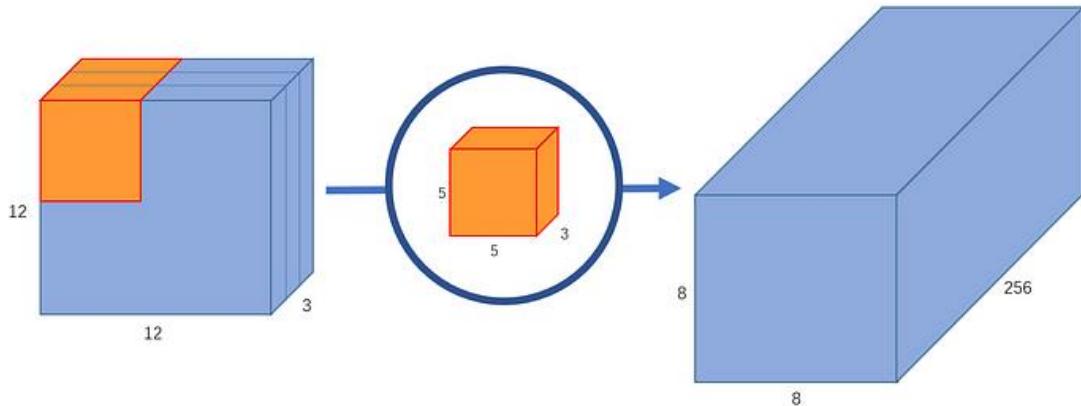
However, because the image has 3 channels, our convolutional kernel needs to have 3 channels as well. This means, instead of doing  $5 \times 5 = 25$  multiplications, we actually do  $5 \times 5 \times 3 = 75$  multiplications every time the kernel moves.

Just like the 2-D interpretation, we do scalar matrix multiplication on every 25 pixels, outputting 1 number. After going through a 5x5x3 kernel, the 12x12x3 image will become a 8x8x1 image.



What if we want to increase the number of channels in our output image? What if we want an output of size 8x8x256?

Well, we can create 256 kernels to create 256 8x8x1 images, then stack them up together to create a 8x8x256 image output.



This is how a normal convolution works. I like to think of it like a function: 12x12x3 — (5x5x3x256) — >12x12x256 (Where 5x5x3x256 represents the height, width, number of input channels, and number of output channels of the kernel). Note that this is not matrix multiplication; we're not multiplying the whole image by the kernel but moving the kernel through every part of the image and multiplying small parts of it separately.

A depthwise separable convolution separates this process into 2 parts: a depthwise convolution and a pointwise convolution.

#### Depthwise Convolution

In the first part, depthwise convolution, we give the input image a convolution without changing the depth. We do so by using 3 kernels of shape 5x5x1.

Each 5x5x1 kernel iterates 1 channel of the image (note: 1 channel, not all channels), getting the scalar products of every 25 pixel group, giving out a 8x8x1 image. Stacking these images together creates a 8x8x3 image.

#### Pointwise Convolution

Remember, the original convolution transformed a 12x12x3 image to a 8x8x256 image.

Currently, the depthwise convolution has transformed the 12x12x3 image to a 8x8x3 image.

Now, we need to increase the number of channels of each image.

The pointwise convolution is so named because it uses a 1x1 kernel, or a kernel that iterates through every single point. This kernel has a depth of however many channels the input image has; in our case, 3. Therefore, we iterate a 1x1x3 kernel through our 8x8x3 image, to get a 8x8x1 image.

We can create 256 1x1x3 kernels that output a 8x8x1 image each to get a final image of shape 8x8x256.

And that's it! We've separated the convolution into 2: a depthwise convolution and a pointwise convolution. In a more abstract way, if the original convolution function is  $12 \times 12 \times 3$  —  $(5 \times 5 \times 3 \times 256) \rightarrow 12 \times 12 \times 256$ , we can illustrate this new convolution as  $12 \times 12 \times 3 - (5 \times 5 \times 1 \times 1) -> (1 \times 1 \times 3 \times 256) -> 12 \times 12 \times 256$ .

### Main Result

Let's calculate the number of multiplications the computer has to do in the original convolution.

There are 256  $5 \times 5 \times 3$  kernels that move 8x8 times. That's  $256 \times 3 \times 5 \times 5 \times 8 \times 8 = 1,228,800$  multiplications.

What about the separable convolution? In the depthwise convolution, we have 3  $5 \times 5 \times 1$  kernels that move 8x8 times. That's  $3 \times 5 \times 5 \times 8 \times 8 = 4,800$  multiplications. In the pointwise convolution, we have 256  $1 \times 1 \times 3$  kernels that move 8x8 times. That's  $256 \times 1 \times 1 \times 3 \times 8 \times 8 = 49,152$  multiplications. Adding them up together, that's 53,952 multiplications.

53,952 is a lot less than 1,228,800. With less computations, the network is able to process more in a shorter amount of time.

How does that work, though? The first time I came across this explanation, it didn't really make sense to me intuitively. Aren't the two convolutions doing the same thing? In both cases, we pass the image through a  $5 \times 5$  kernel, shrink it down to one channel, then expand it to 256 channels. How come one is more than twice as fast as the other?

After pondering about it for some time, I realized that the main difference is this: in the normal convolution, we are transforming the image 256 times. And every transformation uses up  $5 \times 5 \times 3 \times 8 \times 8 = 4800$  multiplications. In the separable convolution, we only really transform the image once — in the depthwise convolution. Then, we take the transformed image and simply elongate it to 256 channels. Without having to transform the image over and over again, we can save up on computational power.

It's worth noting that in both Keras and Tensorflow, there is a argument called the "depth multiplier". It is set to 1 at default. By changing this argument, we can change the number of output channels in the depthwise convolution. For example, if we set the depth multiplier to 2, each  $5 \times 5 \times 1$  kernel will give out an output image of  $8 \times 8 \times 2$ , making the total (stacked) output of the depthwise convolution  $8 \times 8 \times 6$  instead of  $8 \times 8 \times 3$ . Some may choose to manually set the depth multiplier to increase the number of parameters in their neural net for it to better learn more traits.

Are the disadvantages to a depthwise separable convolution? Definitely! Because it reduces the number of parameters in a convolution, if your network is already small, you might end up with too few parameters and your network might fail to properly learn during training. If used properly, however, it manages to enhance efficiency without significantly reducing effectiveness, which makes it a quite popular choice.

### 1x1 Kernels

A 1x1 kernel — or rather, n 1x1xm kernels where n is the number of output channels and m is the number of input channels — can be used outside of separable convolutions. One obvious purpose of a 1x1 kernel is to increase or reduce the depth of an image. If you find that your convolution has too many or too little channels, a 1x1 kernel can help balance it out.

For me, however, the main purpose of a 1x1 kernel is to apply non-linearity. After every layer of a neural network, we can apply an activation layer. Whether it be ReLU, PReLU, Softmax, or another, activation layers are non-linear, unlike convolution layers. “A linear combination of lines is still a line.” Non-linear layers expand the possibilities for the model, as is what generally makes a “deep” network better than a “wide” network. In order to increase the number of non-linear layers without significantly increasing the number of parameters and computations, we can apply a 1x1 kernel and add an activation layer after it. This helps give the network an added layer of depth.

## Squeeze-and-Excitation Networks

Squeeze-and-Excitation Networks ([SENets](#)) introduce a building block for CNNs that improves channel interdependencies at almost no computational cost. They were used at this year’s ImageNet competition and helped to improve the result from last year by 25%. Besides this huge performance boost, they can be easily added to existing architectures. The main idea is this:

- Let’s add parameters to each channel of a convolutional block so that the network can adaptively adjust the weighting of each feature map.

As simple as it may sound, this is it. So, let’s take a closer look at why this works so well and how we can potentially improve any model with five simple lines of code.

### THE “WHY”

CNNs use their convolutional filters to extract hierarchical information from images. Lower layers find trivial pieces of context like edges or high frequencies, while upper layers can detect faces, text or other complex geometrical shapes. They extract whatever is necessary to solve a task efficiently.

All of this works by **fusing the spatial and channel information of an image**. The different filters will first find spatial features in each input channel before adding the information across all available output channels.

All you need to understand for now is that **the network weights each of its channels equally when creating the output feature maps**. SENets are all about changing this by adding a content aware mechanism to weight each channel adaptively. In its most basic form this could mean adding a single parameter to each channel and giving it a linear scalar how relevant each one is.

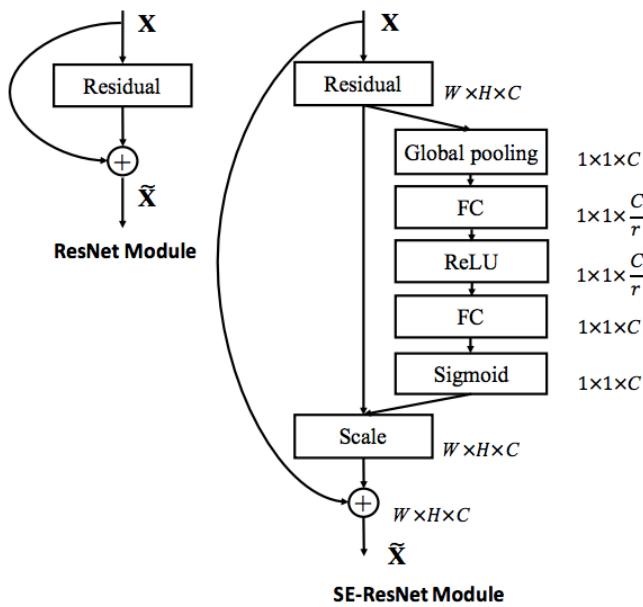
However, the authors push it a little further. First, they get a global understanding of each channel by squeezing the feature maps to a single numeric value. This results in a vector of size n, where n is equal to the number of convolutional channels. Afterwards, it is fed through a two-

layer neural network, which outputs a vector of the same size. These  $n$  values can now be used as weights on the original features maps, scaling each channel based on its importance.

### THE “HOW”

1. The function is given an input convolutional block and the current number of channels it has.
2. We squeeze each channel to a single numeric value using average pooling.
3. A fully connected layer followed by a ReLU function adds the necessary nonlinearity. It's output channel complexity is also reduced by a certain ratio.
4. A second fully connected layer followed by a Sigmoid activation gives each channel a smooth gating function.
5. At last, we weight each feature map of the convolutional block based on the result of our side network.

These five steps add almost no additional computing cost (less than 1%) and can be added to any model.



The authors show that by adding SE-blocks to ResNet-50 you can expect almost the same accuracy as ResNet-101 delivers. This is impressive for a model requiring only half of the computational costs. The paper further investigates other architectures like Inception, Inception-ResNet and ResNeXt. The latter leads them to a modified version that shows a top-5 error of 3.79% on ImageNet.

	original		re-implementation			SENet		
	top-1 err.	top-5 err.	top-1 err.	top-5 err.	GFLOPs	top-1 err.	top-5 err.	GFLOPs
ResNet-50 [9]	24.7	7.8	24.80	7.48	3.86	23.29 <sub>(1.51)</sub>	6.62 <sub>(0.86)</sub>	3.87
ResNet-101 [9]	23.6	7.1	23.17	6.52	7.58	22.38 <sub>(0.79)</sub>	6.07 <sub>(0.45)</sub>	7.60
ResNet-152 [9]	23.0	6.7	22.42	6.34	11.30	21.57 <sub>(0.85)</sub>	5.73 <sub>(0.61)</sub>	11.32
ResNeXt-50 [43]	22.2	-	22.11	5.90	4.24	21.10 <sub>(1.01)</sub>	5.49 <sub>(0.41)</sub>	4.25
ResNeXt-101 [43]	21.2	5.6	21.18	5.57	7.99	20.70 <sub>(0.48)</sub>	5.01 <sub>(0.56)</sub>	8.00
BN-Inception [14]	25.2	7.82	25.38	7.89	2.03	24.23 <sub>(1.15)</sub>	7.14 <sub>(0.75)</sub>	2.04
Inception-ResNet-v2 [38]	19.9 <sup>†</sup>	4.9 <sup>†</sup>	20.37	5.21	11.75	19.80 <sub>(0.57)</sub>	4.79 <sub>(0.42)</sub>	11.76

### **Region-Based Convolutional Neural Networks (R-CNN) Family:**

Now, we're going to the next page. From image classification to object detection. So far, what we've discussed was one large convolutional neural network. Although they have many layers, they are one network at the end. But **R-CNN** and its variants that will be our focus, have this network as a part. A **CNN** is now a part of an entire “system” processing more complex tasks such as **object detection** and **image segmentation**. Starting with R-CNN, we're going to see how these networks had been transformed and the underlying idea for the changes.

#### **From image classification to object detection:**

Before we jump right into the R-CNN “family,” let's briefly check the basic idea of image classification and image detection. What's the difference between them? What additional work do we need to do for detecting objects in an image?

First, whatever the number of classes we have, there is going to be an additional class- the background. An object detector is required to answer the question, “is there an object?”, which is not the case for image classification. Second, when there is an object, it's still not sufficient with “yes, there is!” (Quite ridiculous to imagine.. 😅) It should also tell us, “where is the object located?” We need the **position of the detected objects**. This may sound simple but the implementation isn't. And when we take speed and efficiency challenges into account, things get even more complicated.

Therefore object detection would be like.. looking for the regions of an object, localizing it and classifying what it is. Having this basic concept in mind, we're now ready to start the second topic from R-CNN.



## R-CNN

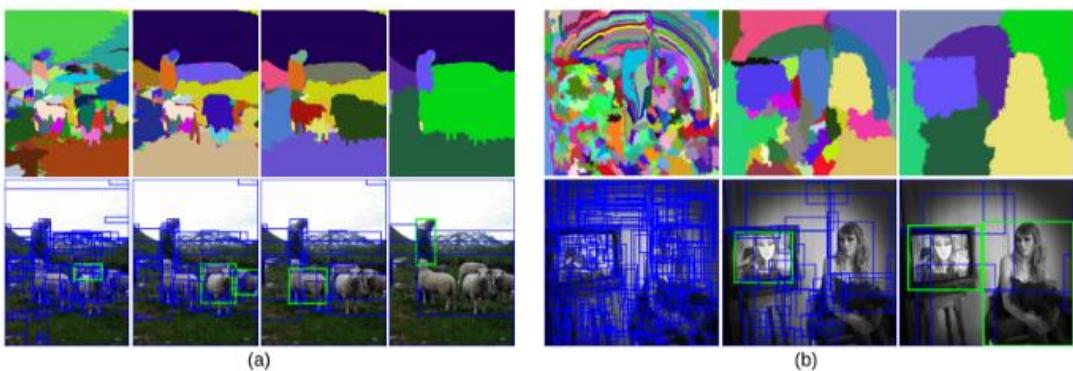
R-CNN answers the question: “**to what extent can the CNN classification results on ImageNet generalize to object detection results?**” So this network can be said to be the beginning of **object detection family tree**, which has great importance in the application of a neural network. The basic structure is composed of three steps-**extracting region proposals, computing CNN, and classification**.

First, we extract some regions that seem promising to have an object from an input image. **R-CNN** used **selective search** for getting those **regions of interest (ROI)**. **Selective search** is a **region**

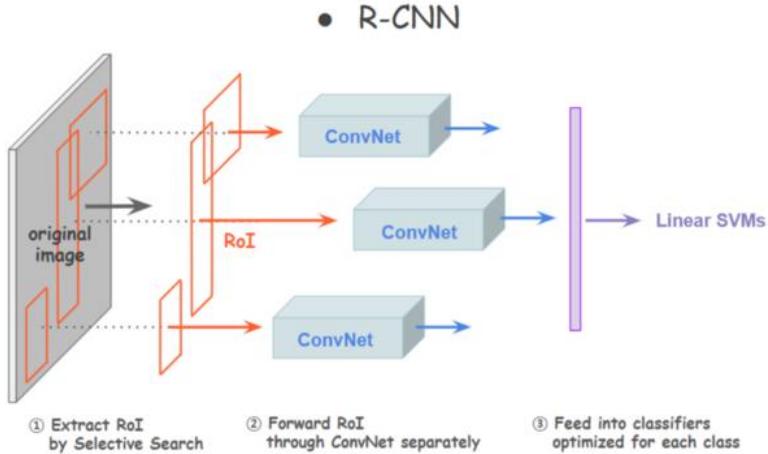
**proposal algorithm that segments an image based on the intensity of the pixels.** It is a classic computer vision technique that is used to propose candidate regions or bounding boxes of potential objects in the image, works in 3 stages:

- Stage 1 – Calculate initial regions.
- Stage 2 – Group regions with the highest similarity – and repeat.
- Stage 3 – Generate a hierarchy of bounding boxes.

If you'd like to get an in-depth understanding of Selective Search, here is [the original paper](#). The basic idea is shown below. It first starts with the **overly segmented picture** and draws a **bounding box** around each segment. And based on their similarities in color, texture, size and shape compatibility, it keeps grouping adjacent ones and forming larger segments. R-CNN extracts around 2000 region proposals by this method and feeds them to the CNN. And this is the reason it is named R-CNN, **Regions with CNN features**.

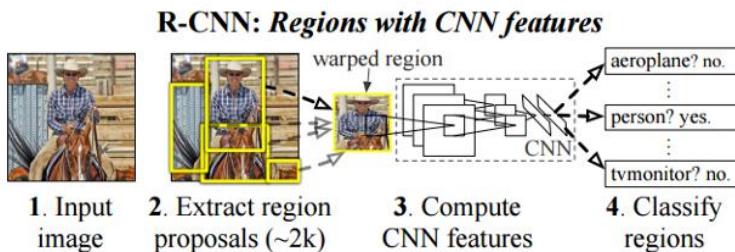


After getting the candidates, the second step is entering a large convolutional neural network. **Each proposal is resized into a fixed size and is input into the CNN separately.** R-CNN used AlexNet (It was 2014, there weren't ResNet or InceptionNet at that time) and **we get 4096-dimensional feature vector from each proposal.** To solve the problem that the input of the CNN should be a fixed size ( $224 \times 224$ ) but the size of each proposal is different (the size of the objects must be changed to a fixed size), they used warping – anisotropically scale (different scale) the object proposals (this is done using **Spatial Pyramid Pooling - SPP**).



And at the last step, the output extracted from the CNN is fed into a set of **class-specific linear SVM models**. We optimize one linear SVM per each class, and we get the output image with the bounding boxes whose score is higher than a threshold value. In case of having overlapping boxes, it only takes one by applying **non-maximum suppression**.

One interesting part worth mentioning is how it overcame the data scarcity issue. The researchers had the challenge to **train such a large network with only a small quantity of labeled data**. And the solution was **pre-training the CNN on a different data with labels** (which is supervised learning), **and then do fine-tuning with the original dataset**.



You may have noticed some inefficient parts that could be enhanced. The **selective search is computation intensive**. And **processing the CNN for each ROI was repetitive work**, which **requires a huge amount of computation cost** again. The need for the pre-training process and the separate classifiers were unattractive. And the storage for these models is too big. Although R-CNN was a milestone achievement, it had several drawbacks to be improved.

### SPATIAL PYRAMID POOLING (SPP)

**The input image of a CNN should have a fixed size.** In order to give an image of arbitrary size as input to a CNN, we should resize the image either by cropping or warping. In the image below, we can see an example of these two resizing techniques:



We can easily observe some problems with resizing the input image. Specifically:

- When we **crop** the input image, we run the **risk of removing visual information that is useful for the recognition task**. For example, in the above image, the front and the back part of the car are missing after cropping. These parts of the car are important in recognition.
- When we **warp** the image, we **change the geometry of the objects in the image resulting in unwanted distortion**.

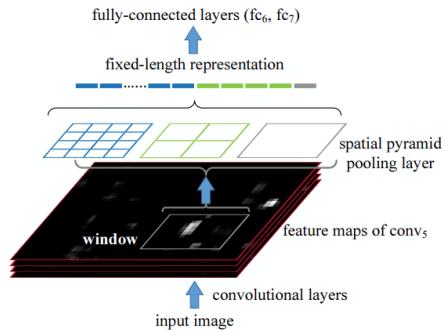
As a result of the previous problems, there is a decrease in the performance of CNN models when they take input images with variable input sizes.

### Why do CNN architectures require inputs with fixed input sizes?

In fully-connected layers you train weight to connect every dimension of the input with every dimension of the output, so if you made the input larger, you would require more weights. But you cannot just make up new weights, they would need to be trained. So, for fully-connected layers the weight matrix determines both input and output size.

The solution to this problem lies in the **Spatial Pyramid Pooling (SPP)** layer. It is placed between the last Conv layer and the first FC layer and removes the fixed-size constraint of the network.

The goal of the SPP layer is to pool the variable-size features that come from the Conv layer and generate fixed-length outputs that will then be fed to the first FC layer of the network. Below, we can see how the SPP layer works:



The input of the SPP layer is a set of  $d$  feature maps of arbitrary size. Then, the SPP layer maintains spatial information of these feature maps in local spatial bins. The output size of the SPP layer is fixed because the number and the size of these bins are always fixed. Specifically, we have three kinds of bins that act as pooling layers:

1. The **first pooling layer** (right in the image) consists of a **single bin** that outputs a  $d$ -dimensional feature since it applies **max pooling in each input feature map**.
2. The **second pooling layer** (middle in the image) consists of **4 bins** that output a  $4 \times d$ -dimensional feature.
3. The **third pooling layer** (left in the image) consists of **16 bins** that output a  $16 \times d$ -dimensional feature.

**All pooling layers apply max pooling.** In total, the output feature has a fixed size that is equal to  $(16 + 4 + 1) \times d$ .

## TRAINING R-CNN

### *Training of the CNN*

R-CNNs uses pre-trained CNNs, usually on pre-trained image-level (not object level) annotations. Adapting the CNN to detection and to the new domain (warped proposal windows), by training with SGD (Stochastic Gradient Descent). It uses proposals with  $IoU > 0.5$  as positive examples (the rest are negative). Recall that this is a supervised task, and thus we have the ground-truth bounding boxes.

### *Training of the SVM*

Uses proposals with  $IoU < 0.3$  as negative examples and ground truth regions as positive examples. Why is it different than the CNN threshold? It empirically works better and reduces overfitting.

## DETECTING OBJECTS WITH R-CNN

**Non-Maximum suppression (NMS)** is applied greedily given all scored regions in an image. Non-Maximum Suppression (NMS) is a technique which filters the proposals based on some threshold value (rejects proposals).

- **Input:** A list of Proposal boxes  $B$ , corresponding confidence scores  $S$  and overlap threshold  $N$ .
- **Output:** A list of filtered proposals  $D$ .



## DRAWBACKS OF R-CNN

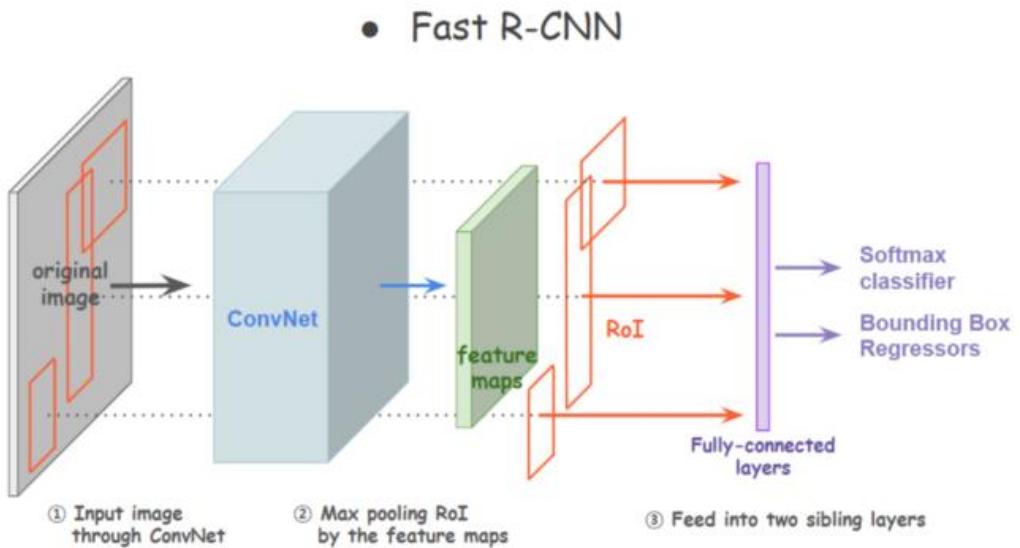
- Training is not end-to-end, but a multi-stage pipeline.
- Training is computationally expensive in space and time (training a deep CNN on so many region proposals per image is very slow).
- At test-time object-detection is slow, requiring a CNN-based feature extraction to pass on each of the candidate regions generated by the region proposal algorithm.

## Fast R-CNN

**Fast R-CNN** is the next version of the previous work. What changes had been made here?

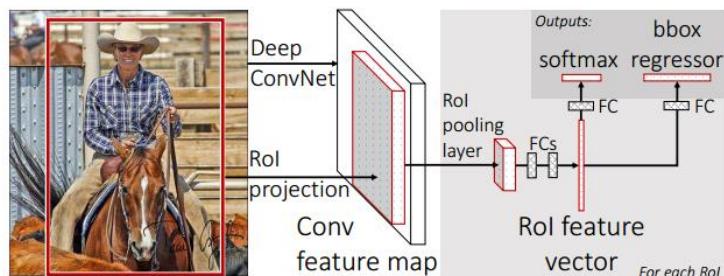
**Processing convolutional mapping repetitively was improved.** The first change occurred at the repetitive convolution layers.

Let's assume it takes  $N$  seconds to compute one single convolutional network. As R-CNN input 2000 RoI to the network separately, the total processing time will be  $2000 \cdot N$  seconds. Instead of processing the CNN individually, now we do it only once (**single stage**) by sharing the convolution with the proposals altogether.



As you can see above, this network takes two data input, an original image and a set of the region proposals as input. The whole image feeds forward through the network to produce a **feature map**. With this feature map, each region proposal passes a pooling layer and fully connected layers to create a **feature vector**. Therefore, the convolutional computation is done once, not for each proposal.

And the multiple classifiers are replaced with two sibling layers. One is a **softmax function for classifying the object** with the possibility estimates for each class, and the other is a **bounding-box regressor returning the coordinates of the detected object**. So the resulting feature vector is fed into these two layers and we get the outcome from the two layers.



Now, this model made changes to share convolutions and to detach the additional classifiers. By merging “multiple bodies” into one and taking off the “heavy tail,” we could achieve less computation and storage. This architecture allows us to train all the weights together, including even that of the softmax classifier and the multiple regressors. And this means the propagation will flow back and forth updating all the weights. Awesome progress! However, we still have a chance to get better performance.

## ROI POOLING LAYERS

First of all, in the Fast R-CNN architecture a Fully Connected Layer, with a fixed size follows the RoI pooling layer. Therefore, because the RoI windows are of different sizes, a pooling technique needs to be applied to them.

The RoI pooling layer, a **Spatial pyramid Pooling (SPP) technique** is the main idea behind Fast R-CNN and the reason that it outperforms R-CNN in accuracy and speed respectively. SPP is a pooling layer method that aggregates information between a convolutional and a fully connected layer and cuts out the fixed-size limitations of the network. Therefore, the disadvantage of the fixed image size needs is overpassed.

In general, an RoI is a rectangular window and is defined by  $(r, c)$  and  $(h, w)$  values that correspond to each window’s top-left corner, height, and width respectively. RoI pooling layers divide a  $(h, w)$  rectangular window into  $H \times W$  set of sub-windows, and afterward perform max pooling in each sub-window. The RoI pooling layer performs a max pooling operation in any proposed RoI of an image individually.

Note that the authors of Fast R-CNN propose the value of 7 for  $H$  and  $W$ .

## TRUNCATED SVD

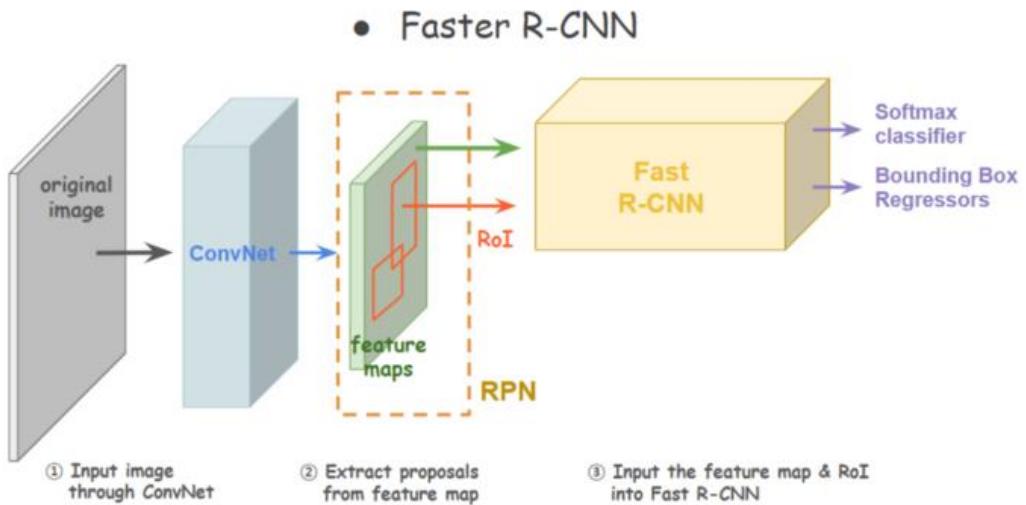
When it comes to matrix factorization technique, **Truncated Singular Value Decomposition (SVD)** is a popular method to produce features that factors a matrix  $M$  into the three matrices  $U$ ,  $\Sigma$ , and  $V$ . Another popular method is **Principal Component Analysis (PCA)**. Truncated SVD shares similarity with PCA while SVD is produced from the data matrix and the factorization of PCA is generated from the covariance matrix. Unlike regular SVDs, **truncated SVD produces a factorization where the number of columns can be specified for a number of truncations**. For example, given an  $n \times n$  matrix, truncated SVD generates the matrices with the specified number of columns, whereas SVD outputs  $n$  columns of matrices.

### *The advantages of truncated SVD over PCA*

Truncated SVD can deal with sparse matrix to generate features’ matrices, whereas PCA would operate on the entire matrix for the output of the covariance matrix.

## FASTER R-CNN

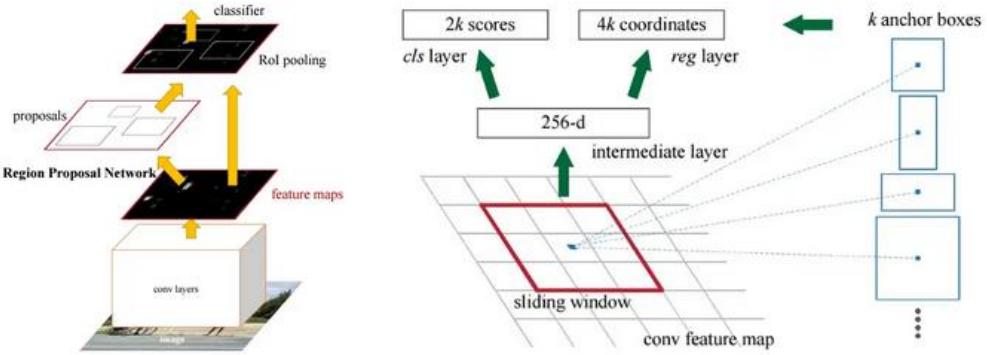
Still, Fast R-CNN was hard to be used in real time detection. And the major reason for such time delay stemmed from **selective search**. It had computation bottleneck and it needed to be replaced with a more efficient way. But how? **How could we get region proposals without Selective Search?** What about maximizing the usage of the ConvNet? Researchers found that the feature maps in Fast R-CNN also can be used for **generating region proposals**. Therefore by being free from Selective Search, much efficient network could be developed.



**Faster R-CNN** is composed of two modules, **Region Proposal Network (RPN)** and **Fast R-CNN**. We first input an image to a “mini-network” and it will output the feature maps. By sliding a small window over the feature maps, we extract region proposals. And each proposal is fed into two sibling layers, a softmax classifier and a bounding-box regressor.

These two layers may seem similar to the last layers of Fast R-CNN, but they are derived for a different purpose. For each proposal, the classifier estimates **the probability of the presence of an object in an image**. And the regressor returns **the coordinates of the bounding box**. So they are for generating candidates, not predicting the actual objects.

We only take the proposals whose scores are higher than a threshold and input them with the feature map together to Fast R-CNN. The following steps are the same. They are input into a convolutional network and RoI pooling layers. And then the last layers will be a classifier and a regressor that finally predicts the real objects in images.



One important property of this network is **translation invariant**, which is achieved by **anchors** and the way it computes proposals relative to the anchors. What is translation invariant? Simply put, it means we can detect an object regardless of its rotation, relocation, size change and what not. An object in an image can be at the center or the upper left. The same object can be at wide or long scale depending on the perspective. To prevent a model from failing to locate an object because of translation, we **make anchor boxes of multiple scales and aspect ratios** as in the picture shown above. These boxes are put at the center of the sliding window. So if there's  $K$  number of the boxes at a certain position, we get  $2K$  scores and  $4K$  coordinates of  $K$  boxes.

In conclusion, at the **Region Proposal Network (RPN)**, we **slide a window with multiple anchor boxes over the feature map and evaluate each box by the classifier and the regressor**. The proposals below the threshold are rejected, hence, feeding only the promising one to the next steps.

More than that, this model optimized in 4 steps of alternative training by fine-tuning the layers unique to RPN and Fast R-CNN separately while fixing the shared layers. This allows the model to share weights forming a unified network and brought higher performance both in efficiency and accuracy. To compare the performance, The time for proposals of Faster R-CNN was 10 milliseconds per image (5 frames per second for the whole process), while 2 seconds for Selective Search using CPU.

### REGION PROPOSAL NETWORK (RPN)

An RPN is a **fully-convolutional network** that simultaneously predicts **object bound** and **objectness scores** at each position. RPN and Fast R-CNN can be trained using a simple alternating optimization to share convolutional features. It works in several steps:

- Generate anchor boxes (different sizes and different aspect ratios).
- Feed the possible regions into the RPN and classify each anchor box whether it is foreground or background (anchors that have a higher overlap with ground-truth boxes should be labeled as foreground, while others should be labeled as background).
- The output is fed into a Softmax or logistic regression activation function, to predict the labels for each anchor. A similar process is used to refine the anchors and define the bounding boxes for the selected features (learn the shape offsets for anchor boxes to fit them for objects).

Now the next networks to be explored are **SSD**, **YOLO**, **FPN** and **RetinaNet**. These networks are based on **one-stage architecture**. The biggest change is that these **no longer need regional proposals**, hence forming a more unified and fully convolutional network, which results in brings **faster performance**. Now let's see how this can be possible one by one.

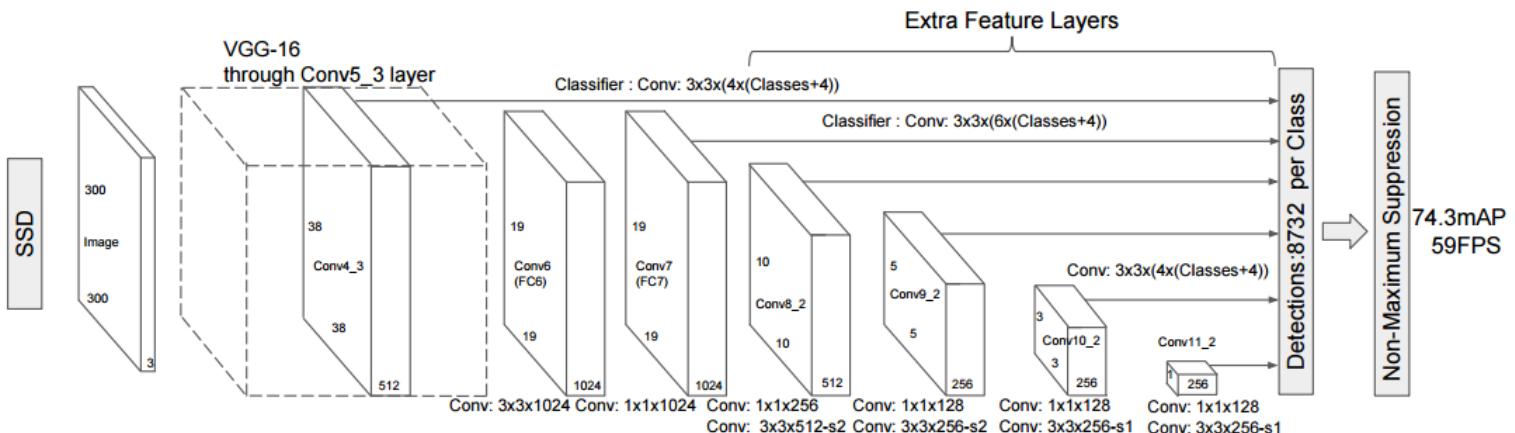
## Single Shot Multibox Detector (SSD)

**SSD** stands for **Single Shot Detector**, and there are three main points in its architecture. First, this **network can detect in multiple scales**. With the layers coming after the base network, it **yields several feature maps with different resolutions**, which enables the network to **work at multi-scales**.

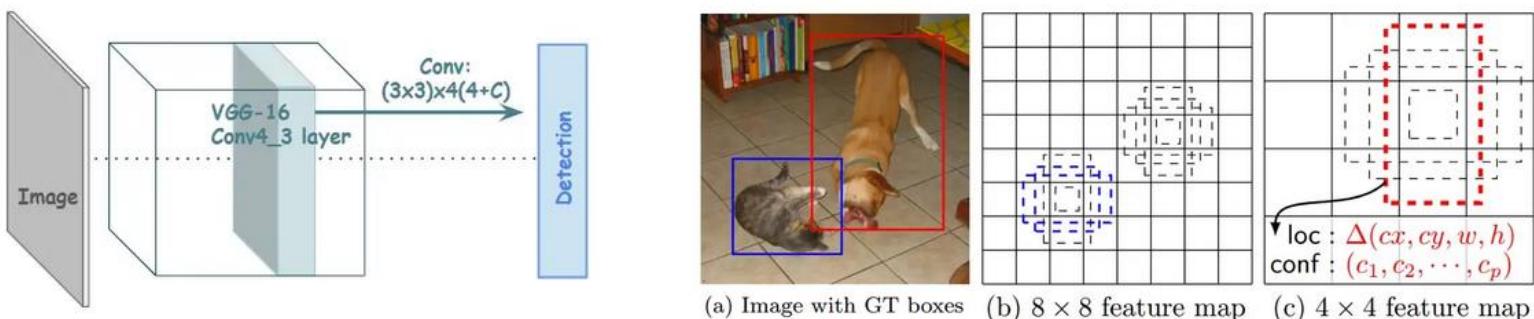
Second, these **predictions are made in a convolutional way**. A set of convolutional filters **make the predictions for the location of the objects and the class score**. This is where the name of "**Single Shot**" comes from. **Instead of having additional classifiers and regressors, now the detections are made in a single shot!**

And lastly, there are a **fixed number of default boxes associated with these feature maps**.

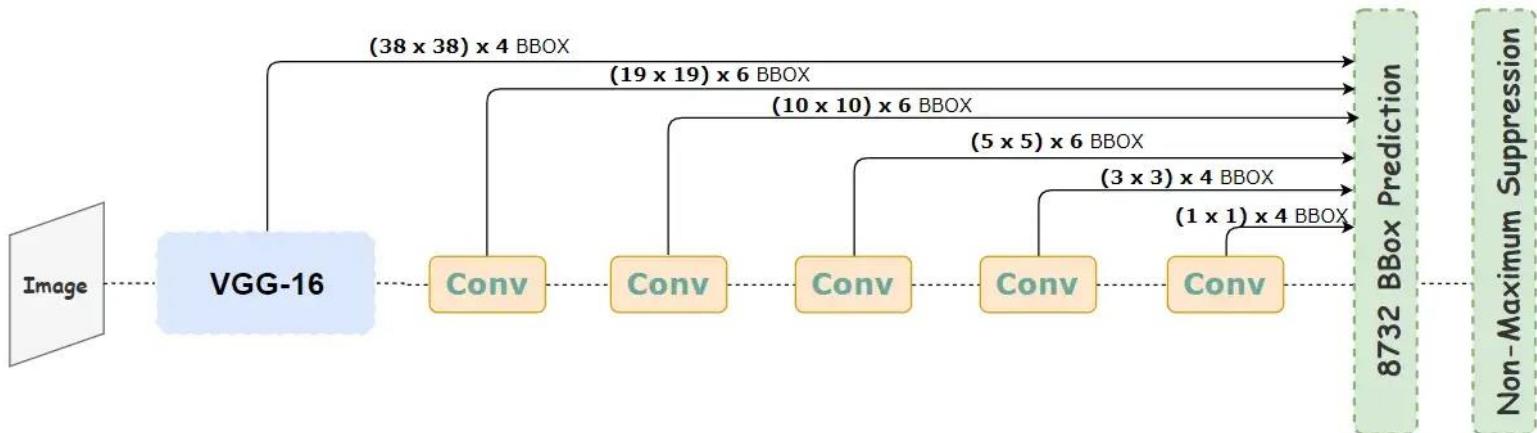
Similar to the anchors of Faster R-CNN, the default boxes are applied at each feature map cell.



The entire architecture of the network is as shown above. It uses VGG-16 as its base network (**backbone**), and there are additional feature layers which decrease in size progressively (**SSD head**). This can be considered to be quite complicated at first glance, so let's split it into two parts.



In the picture on the left, the **feature layer from the base network produces a detection by the convolutional filters**. When the dimensional size of the feature layer is  $N \times N$  with  $P$  channels, the filter size here is  $3 \times 3 \times P$ . Now, as I said above, there are a **set of default bounding boxes at each cell**. So if we have  $K$  number of the boxes and  $C$  number of classes (including the “background” class), we compute the four offset values against the original default box, and the class scores for each box. Therefore the total number of filters becomes  $(N \times N)K(4 + C)$ .



**SSD makes all the predictions for the bounding box location and the class scores at once** with the convolutional filters. It doesn't require a region proposal network anymore. And the same process is done at the extra feature layers in different scales, which improves accuracy significantly. Recall that the feature maps in different resolutions, see the different sizes of figures or objects in an image. Therefore, the prediction across scales becomes critical in detection tasks.

After we get the outcome from all these layers, we match them with the ground truth box and choose the best one. As most of the boxes are going to be overlapped heavily, we can use **Non-Maximum Suppression** to choose one. But still, there is quite a challenging problem, the **class imbalance**. When we use the regional proposals, they are reasonable candidates for having an object. But here, the network starts from a set of default boxes at each location with multiple scales and shapes. As a result, most of the boxes are going to be negative in the outcome. This creates an imbalance in classes. For example, the “Background” class becomes an easy example and the real “object” classes become hard examples. The easy negative examples can overwhelm the distribution of the classes and degenerate the model.

To remedy this, SSD takes only some of the negative matches so that the ratio between positive and negative matches could be 3:1. This is called **hard negative mining**. Hard negative mining is a classic problem in object detection, which means a **negative class is hard to tell as negative**. A model should classify it as negative, but it gives high confidence as the negative cases become way more than the positive cases. We input only the positive matches into the objective function so that it calculates the loss between the ground truth and the predicted bounding box.

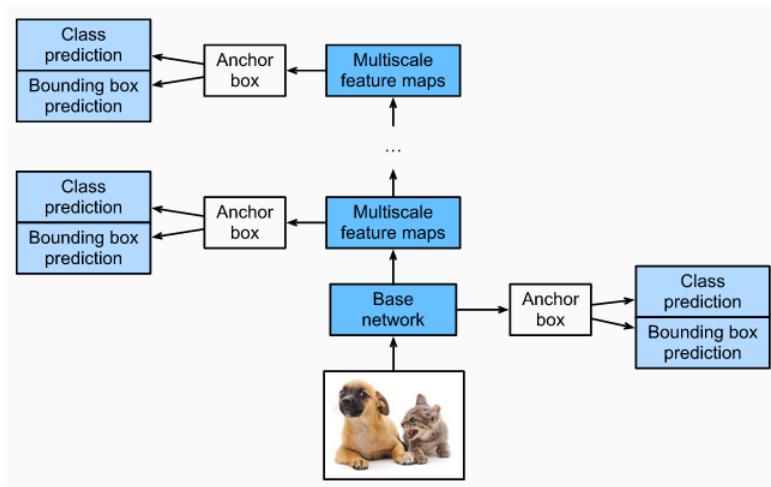
	SSD300			
more data augmentation?	✓	✓	✓	✓
include $\{\frac{1}{2}, 2\}$ box?	✓	✓	✓	✓
include $\{\frac{1}{3}, 3\}$ box?	✓		✓	✓
use atrous?	✓	✓	✓	✓
VOC2007 test mAP	65.5	71.6	73.7	74.2
				74.3

Table 2: Effects of various design choices and components on SSD performance

Prediction source layers from:	mAP		# Boxes
	use boundary boxes? Yes	No	
conv4_3	74.3	63.4	8732
conv7	<b>74.6</b>	63.1	8764
conv8_2	73.8	68.4	8942
conv9_2	70.7	69.2	9864
conv10_2	64.2	64.4	9025
conv11_2	62.4	64.0	8664

Table 3: Effects of using multiple output layers.

One interesting thing is that about 80% of the forward time is spent on the base network, which indicates that a faster base network could even further improve the speed. And data augmentation also plays an important role for higher accuracy. Compared to Faster R-CNN using only an original image and horizontally flipping, variations in input sizes, shapes, and random sampling improve the performance. Also, it is observed that **various default box shapes are better for accuracy** as well.



## ADVANTAGES AND DISADVANTAGES OF THE SSDS

One of the main advantages of SSDs is their **speed and efficiency**. Because they use a single network, they can detect objects in real-time, making them suitable for applications such as self-driving cars and surveillance systems.

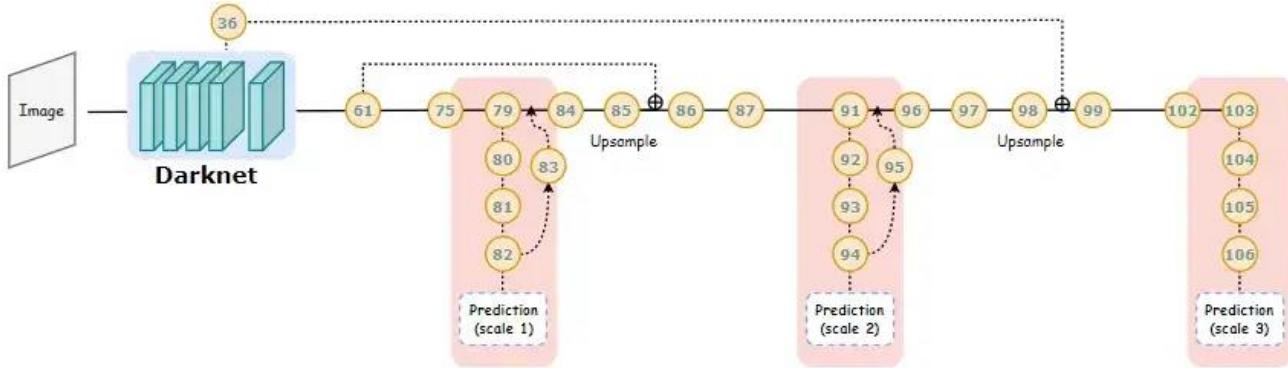
Additionally, because they use a pre-trained base network, SSDs can take advantage of a large amount of labeled data available for image classification tasks. This allows them to achieve **high accuracy** even when trained on relatively small datasets.

On the other hand, SSDs have some limitations. First, they are not as accurate as other methods, such as the R-CNN family of methods. This is because using a single network means **SSDs cannot take advantage of the additional context and information that multiple networks provide**.

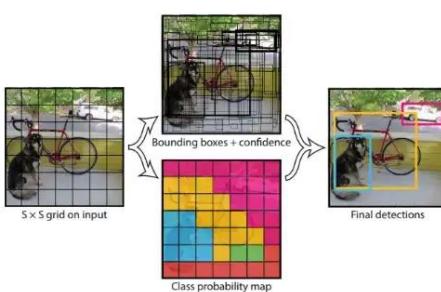
Another limitation of SSDs is that they can be **sensitive to the scale of the objects in an image**. Because the extra layers are designed to detect objects at different scales, SSDs may have difficulty seeing objects significantly smaller or larger than the objects in the training dataset.

## You Only Look Once (Yolo)

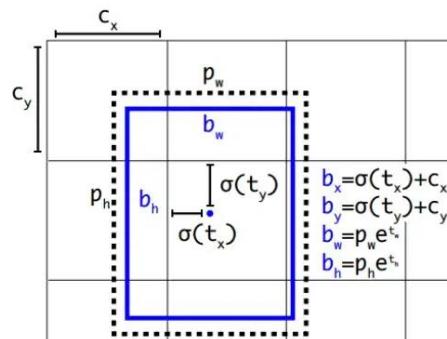
**YOLO** is an acronym of **You Only Look Once**. Similar to SSD, YOLO doesn't use regional proposals and works in a "single-shot" way. There are 3 main variations of YOLO. Rather than covering all the transformation process, I'm going to wrap all and explore the final version. If you'd like to know all the detailed process, check out [this excellent article](#) by Jonathan Hui (Literally, this is the best YOLO tutorial I've ever seen).



The whole network architecture is as shown above. The numbers in the circles are the layer numbers. It uses **Darknet-53** as its base network, and there are additional layers for detecting objects. Let's first put our attention at the tail where the predictions are made. YOLO is also a **fully convolutional network**, so it **predicts the detection by convolution**. After passing through several layers, the detection is made by 1x1 convolution from the feature maps in three different scales.

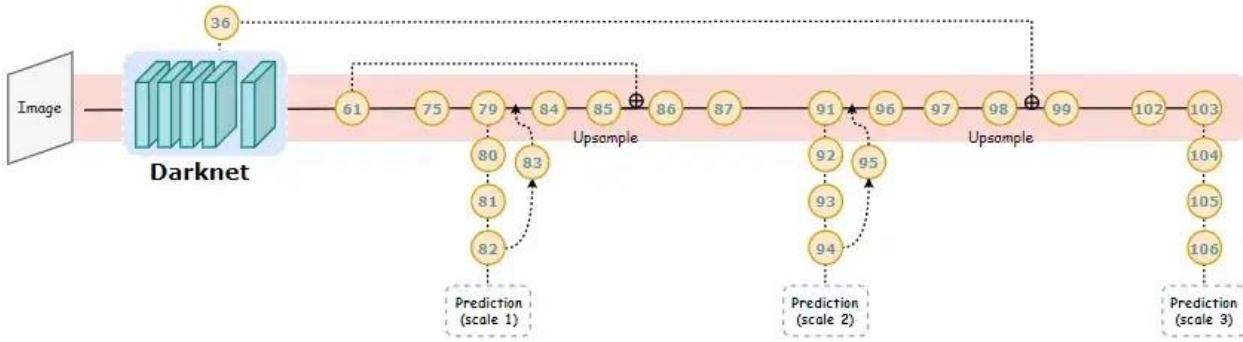


**Figure 2: The Model.** Our system models detection as a regression problem. It divides the image into an  $S \times S$  grid and for each grid cell predicts  $B$  bounding boxes, confidence for those boxes, and  $C$  class probabilities. These predictions are encoded as an  $S \times S \times (B * 5 + C)$  tensor.



**Figure 2. Bounding boxes with dimension priors and location prediction.** We predict the width and height of the box as offsets from cluster centroids. We predict the center coordinates of the box relative to the location of filter application using a sigmoid function. This figure blatantly self-plagiarized from [15].

When the size of the feature map is  $N \times N$  with  $P$  channels,  $1 \times 1 \times P$  size of filters is applied. YOLO also has the default bounding boxes (They are called **anchors** in the original paper). **For each box at each cell, the 4 bounding box offset values, the objectness score and the class scores are predicted.** The objectness score reflects how confident it is that the box contains an object. So now the number of predictions for each grid becomes  $5 + C$ . So if the number of boxes is  $K$ , the total number of filters is  $(N \times N)K(5 + C)$  for each scale.



As I said, **the resolution of the feature maps gets smaller as it goes deeper**, and this gives an impact on the scales when the feature maps see the objects. So to achieve multiple scales, YOLO brings higher resolution features from a previous layer simply by a passthrough layer. This is the same with the shortcut connection of ResNet.

So at the 84th layer, we take the feature map from the two previous layers and upsample it by 2. And then we bring a feature map from the 36th layer and concatenate it with the upsampled features. The same process is repeated one more time at the 96th layer for the final scale. By doing so, we can add finer and more meaningful features into the prediction.

Type	Filters	Size	Output
Convolutional	32	3 x 3	256 x 256
Convolutional	64	3 x 3 / 2	128 x 128
Convolutional	32	1 x 1	
Convolutional	64	3 x 3	
Residual			128 x 128
Convolutional	128	3 x 3 / 2	64 x 64
Convolutional	64	1 x 1	
Convolutional	128	3 x 3	
Residual			64 x 64
Convolutional	256	3 x 3 / 2	32 x 32
Convolutional	128	1 x 1	
Convolutional	256	3 x 3	
Residual			32 x 32
Convolutional	512	3 x 3 / 2	16 x 16
Convolutional	512	1 x 1	
Convolutional	512	3 x 3	
Residual			16 x 16
Convolutional	1024	3 x 3 / 2	8 x 8
Convolutional	512	1 x 1	
Convolutional	1024	3 x 3	
Residual			8 x 8
Avgpool		Global	
Connected		1000	
Softmax			

Table 1: Darknet-53.

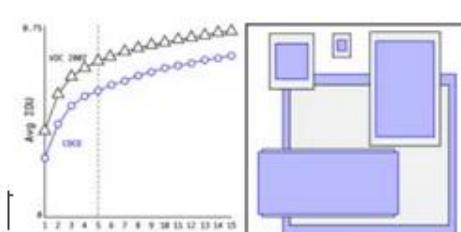


Figure 2: Clustering box dimensions on VOC and COCO. We run k-means clustering on the dimensions of bounding boxes to get good priors for our model. The left image shows the average IOU we get with various choices for  $k$ . We find that  $k = 5$  gives a good tradeoff for recall vs. complexity of the model. The right image shows the relative centroids for VOC and COCO. Both sets of priors favor thinner, taller boxes while COCO has greater variation in size than VOC.

Type	Filters	Size/Stride	Output
Convolutional	32	3 x 3	224 x 224
Maxpool		2 x 2/2	112 x 112
Convolutional	64	3 x 3	112 x 112
Maxpool		2 x 2/2	56 x 56
Convolutional	128	3 x 3	56 x 56
Convolutional	64	1 x 1	56 x 56
Convolutional	128	3 x 3	56 x 56
Maxpool		2 x 2/2	28 x 28
Convolutional	256	3 x 3	28 x 28
Convolutional	128	1 x 1	28 x 28
Convolutional	256	3 x 3	28 x 28
Maxpool		2 x 2/2	14 x 14
Convolutional	512	3 x 3	14 x 14
Convolutional	256	1 x 1	14 x 14
Convolutional	512	3 x 3	14 x 14
Convolutional	256	1 x 1	14 x 14
Convolutional	512	3 x 3	14 x 14
Maxpool		2 x 2/2	7 x 7
Convolutional	1024	3 x 3	7 x 7
Convolutional	512	1 x 1	7 x 7
Convolutional	1024	3 x 3	7 x 7
Convolutional	512	1 x 1	7 x 7
Convolutional	1024	3 x 3	7 x 7
Convolutional	1000	1 x 1	7 x 7
Avgpool		Global	1000
Softmax			

Table 6: Darknet-19.

The base network was AlexNet at ver.1 and Darknet was first used at ver.2 with 19 convolutional layers. This became 53 layers at ver.3 as you can see above. Another important thing worth mentioning is that the default boxes are extracted from **K-means clustering**, rather than being selected by hand. The network learns to adjust the boxes during training but having better priors from the start helps the network for sure. So YOLO ver.3 extracts the 9 default boxes by clustering. By concatenating the higher resolution features from the previous layer and a passthrough layer, YOLO gets better detection with small objects compared to SSD.

## MAIN DIFFERENCE BETWEEN SSD AND YOLO

**Although SSD and YOLO architectures seem to have a lot in common, their main difference lies in how they approach the case of multiple bounding boxes of the same object.** First of all, SSD makes use of fixed-size anchor boxes and takes into consideration the IoU metric (a metric that specifies the amount of overlap between the predicted and ground truth bounding box) with a threshold greater than 0.5.

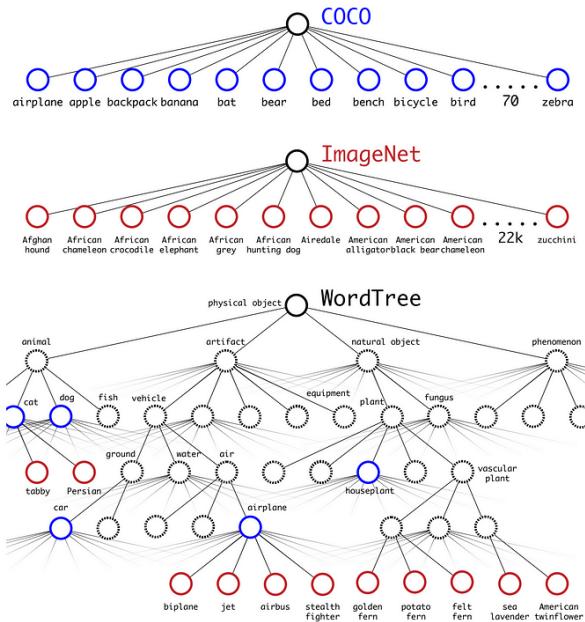
**Moreover, the convolutional model of SSD differs for each feature layer, which is not the case with YOLO's architecture.**

Another difference is that YOLO is limited to predicting bounding boxes since each grid cell is capable of producing predictions for only two boxes and can only belong in one class. **This fact practically affects the network, which cannot handle well images that contain small objects, such as birds, in contrast with SSD.**

As for performance issues, SSD is more accurate in results than YOLO. **Despite this, YOLO is faster and may seem more useful in real-time applications.** One should take into consideration this trade-off of accuracy and speed in order to decide which model is more suitable for his application.

## HIERARCHICAL CLASSIFICATION

YOLO combines labels in different datasets to form a tree-like structure **WordTree**. The children form an is-a relationship with its parent like biplane is a plane. But the merged labels are now not mutually exclusive.



Let's simplify the discussion using the 1000 class ImageNet. Instead of predicting 1000 labels in a flat structure, we create the corresponding WordTree which has 1000 leave nodes for the original labels and 369 nodes for their parent classes. Originally, YOLO predicts the class score for

the biplane. But with the WordTree, it now predicts the score for the biplane given it is an airplane.

$$\text{score}(\text{biplane}|\text{airplane})$$

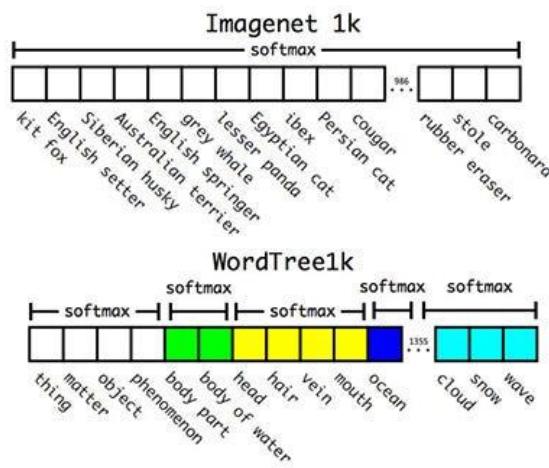
Since:

$$P(\text{biplane}|\text{airplane}) + P(\text{jet}|\text{airplane}) + \dots + P(\text{stealth fighter}|\text{airplane}) = 1$$

we can apply a softmax function to compute the probability:

$$P(\text{class}_i|\text{class}_{\text{parent}})$$

from the scores of its own and the siblings. The difference is, instead of one softmax operations, YOLO performs multiple softmax operations for each parent's children.



The class probability is then computed from the YOLO predictions by going up the WordTree.

$$P(\text{biplane}) = P(\text{biplane}|\text{airplane}) \cdot P(\text{airplane}|\text{air}) \cdot P(\text{air}|\text{vehicle}) \cdot P(\text{physical object})$$

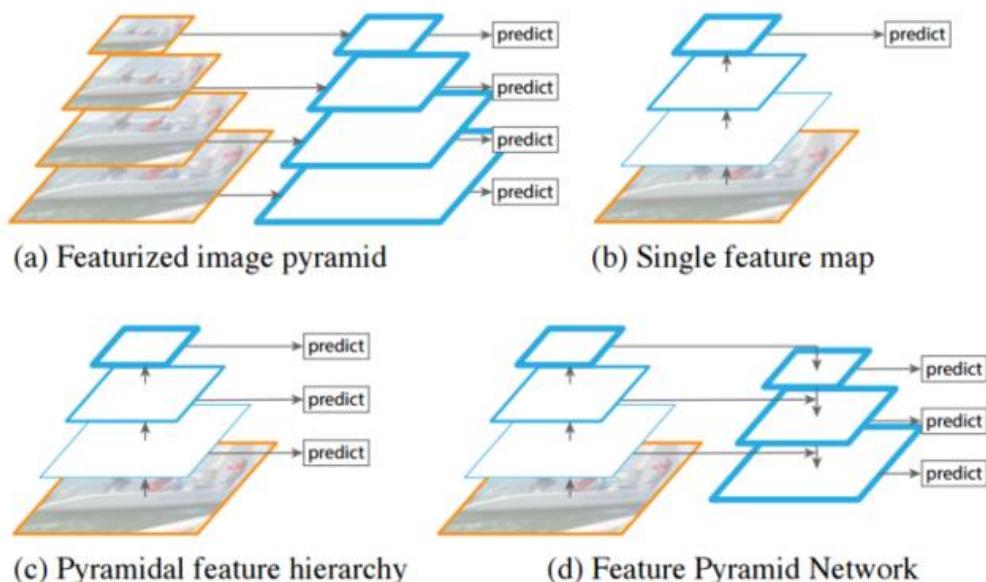
One benefit of the hierarchy classification is that when YOLO cannot distinguish the type of airplane, it gives a high score to the airplane instead of forcing it into one of the sub-categories.

When YOLO sees a classification image, it only backpropagates classification loss to train the classifier. YOLO finds the bounding box that predicts the highest probability for that class and it computes the classification loss as well as those from the parents. (If an object is labeled as a biplane, it is also considered to be labeled as airplane, air, vehicle...) This encourages the model to extract features common to them. So even we have never trained a specific class of objects for object detection, we can still make such predictions by generalizing predictions from related objects.

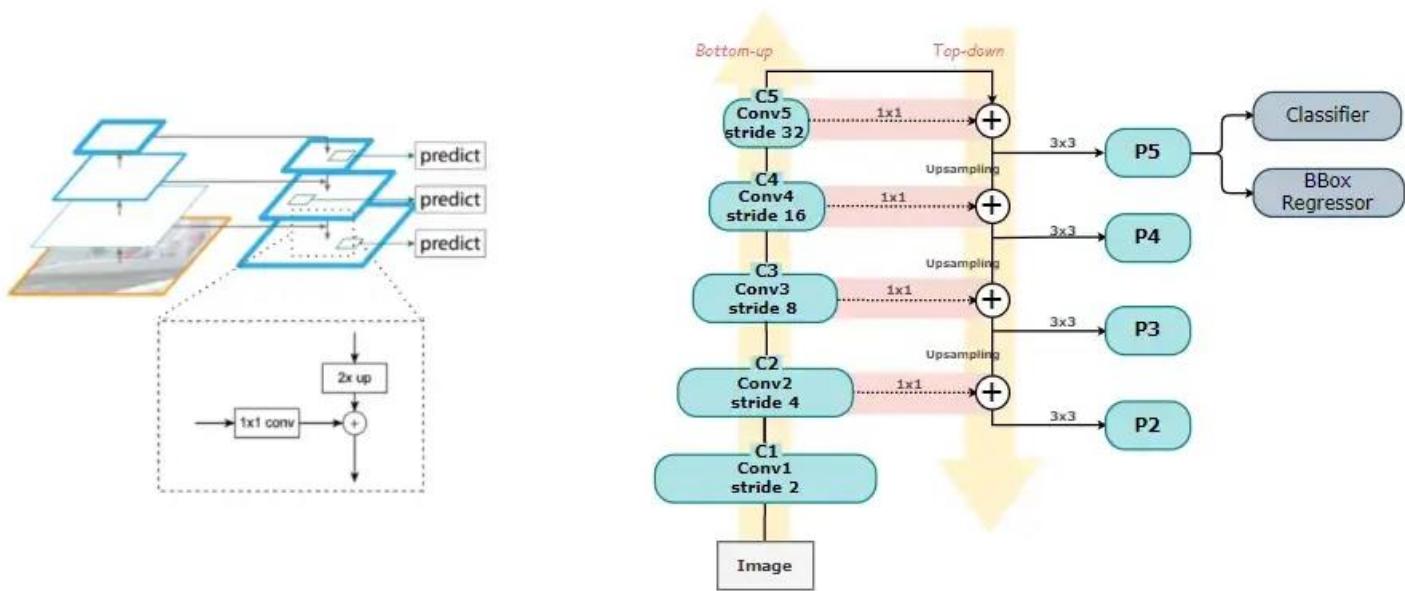
In object detection, we set  $P(\text{physical object})$  equals to the box confidence score which measures whether the box has an object. YOLO traverses down the tree, taking the highest confidence path at every split until it reaches some threshold and YOLO predicts that object class.

## Feature Pyramid Network (FPN)

**FPN** stands for **Feature Pyramid Network**. A network with a pyramid structure has been studied and used before this. And SSD is also a pyramid-like structure (try rotating it vertically). But it can't be said it's a pyramidal feature hierarchy strictly as shown in figure ©. Because it starts to form a pyramid from a higher level, not from the low level. (Remember VGG Conv5\_3 was the first layer for detection)



What's the point having this kind of architecture? **The information for the precise location of an object can be detected in the deeper level of the network**. On the contrary, **the semantics gets stronger in the lower level**. When a network has one way of just going more in-depth, there aren't many semantic data we can use to classify objects at a pixel level. Therefore by



implementing a feature pyramid network, we can produce multi-scale feature maps from all levels, and the features from all these levels are semantically strong.

So the architecture of FPN is as shown above. There are three directions, bottom-up, top-down and lateral directions. The **bottom-up** pathway is the **feedforward computation** of the backbone ConvNet and ResNet as used in the original paper. There are 5 stages and the size of the feature map at each stage has a scaling step of 2. They are differentiated by their dimensions, and we call them as C1, C2, C3, C4 and C5. The higher the layer goes, the smaller the size of the feature maps gets along the bottom-up pathway.

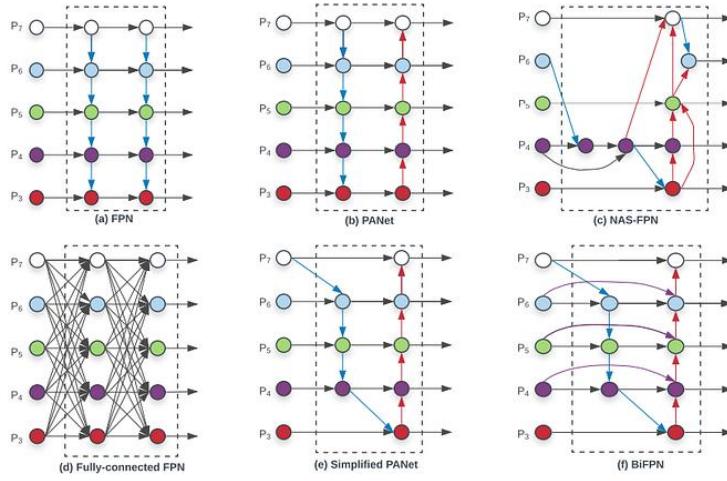
Now at the level C5, we move to the top-down pathway. Upsampling is applied to the output maps at each level. And the corresponding map that has the same spatial size from the bottom-up pathway, is merged via a lateral connection. A 1x1 convolution is applied before the addition to match the depth size. Then, we apply a 3x3 convolution on the merged map to reduce aliasing effect of upsampling. By doing so, we can combine the low resolution and semantically strong features (from top-down pathway) with high resolution and semantically weak features (from bottom-up pathway). This implies we can have rich semantics at all scales. And as shown in the picture, The same process is iterated and produces P2, P3 P4 and P5.

Now the last part of the network is an actual classification. Actually, FPN isn't an object detector in itself. A detector isn't built-in so we need to plug in a detector from this stage. RPN and Fast R-CNN are used in the original paper, and only FPN with RPN case is depicted in the above picture. In the RPN, there are two sibling layers for an object classifier and bounding box regressor. So in FPN, this part is attached to the tail of each level of P2, P3, P4 and P5. I won't cover much about this here, because the reason I picked FPN was to explain RetinaNet.

## EfficientDet

In the EfficientDet paper, two major contributions are made: (a) BiFPN allows bidirectional fast multi-scale feature fusion. (b) a new compound scaling method jointly scales up backbone, feature network, box/class network, and resolution. Using BiFPN and new compound scaling, a family of EfficientDet networks is formed, outperforming previous object detectors in terms of accuracy and efficiency.

## BiFPN



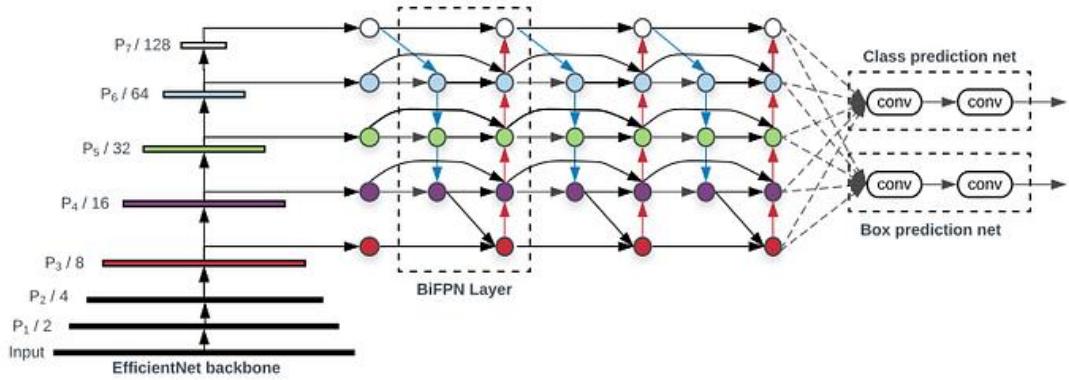
The aim of BiFPN is to effectively aggregate multi-scale features in a top-down manner. The figure above shows the comparison of different FPN based aggregation methods. Starting from (a), a typical FPN network fuses multi-scale features from level 3 to 7. The PANet architecture (b) adds an additional bottom-up pathway on top of FPN to make use of the lower level features efficiently. The connection between the nodes can also be found by using a neural architecture search. As it is shown in (c), the connections are different from the ones constructed by humans.

Comparing the three architectures, the authors observed that PANet achieves better accuracy than FPN and NAS-FPN, but with the cost of more parameters and computations. Starting from the PANet architecture, to improve model efficiency, the authors of the EfficientDet first remove the nodes that only have one input edge and creates the simplified version of the PANet architecture shown in (e). The intuition is that if a node has only one input edge with no feature fusion, then it will have less contribution to the feature network that aims to fuse different features. The authors then add an extra edge from the original input to the output node if the nodes are at the same level. This is done to fuse more features without adding much cost. Lastly, unlike PANet that only has one top-down and one bottom-up path, a bidirectional (top-down & bottom-up) path is treated as one feature network layer and repeats the same layer multiple times to enable more high-level feature fusion. The final proposed BiFPN architecture is shown in (f).

$$\sum_i \frac{w_i}{\varepsilon + \sum_j w_j} I_i$$

A **Weighted Feature Fusion** is utilized when the features are fused together. Since different input features at different resolutions contribute unequally to the output feature, additional weight for each input acts as a scaling factor when the feature fusion is made. Specifically, **Fast normalized fusion** shown in the above equation is used. The positiveness of each  $w$  is ensured by applying relu after each  $w$ , and  $\varepsilon$  is added to avoid numerical instability.

## COMPOUND SCALING



The aim of compound scaling of the network is to scale up a baseline model (EfficientDet D0) to cover a wide spectrum of resource constraints. A simple compound coefficient  $\phi$  jointly scales up all dimensions of the backbone network (BiFPN), class/box network, and resolution. Grid search was used in the EfficientNet paper, but a heuristic-based scaling approach is used because object detectors have much more scaling dimensions than image classification models.

- Backbone network:** The same width/depth scaling coefficients of EfficientNet-B0 to B6 are used to reuse ImageNet-pretrained checkpoints.
- BiFPN network:** Using equation (1), the width of the BiFPN network is grown exponentially and the depth is increased linearly to round the depth value to small integers.

$$W_{bifpn} = 64 \cdot 1.35^\phi, \quad D_{bifpn} = 2 + \phi$$

- Box/class prediction network:** The width of the prediction network is fixed to always be the same as BiFPN, but the depth of the prediction network is linearly increased using the following equation.

$$D_{box} = D_{class} = 3 + \left\lceil \frac{\phi}{3} \right\rceil$$

- Input image resolution:** Since the feature levels 3–7 are used in BiFPN, the input resolution must be divisible by  $2^7 = 128$ , so the resolution of the image is linearly increased using the following equation.

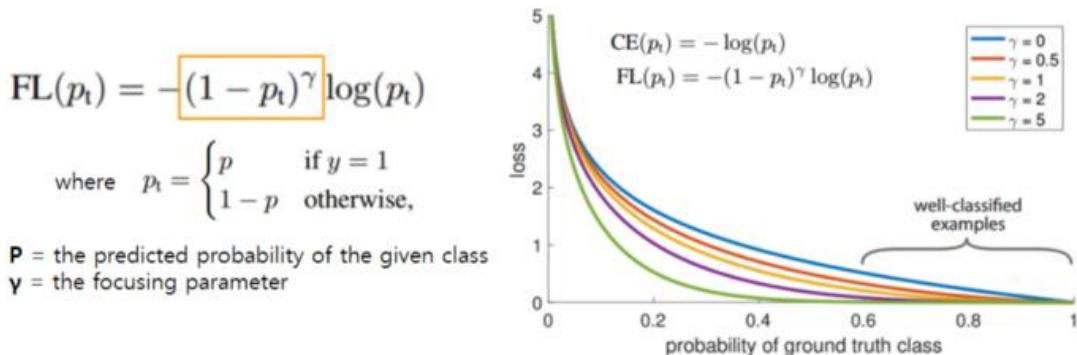
$$R_{input} = 512 + \phi \cdot 128$$

## RESULTS

Using the compound scaling method, a family of EfficientDet networks are generated (EfficientDet D0–07). The table above shows a single-scale performance of the models on the COCO dataset. The EfficientNet models achieve better accuracy and efficiency than previous detectors across a wide range of accuracy or resource constraints. Note that the EfficientDet-D0 model achieves similar accuracy as YOLOv3 with 28x fewer FLOPS. Also, compared to RetinaNet and Mask-RCNN, the EfficientDet-D1 achieves similar accuracy with up to 8x fewer parameters and 25x fewer FLOPS.

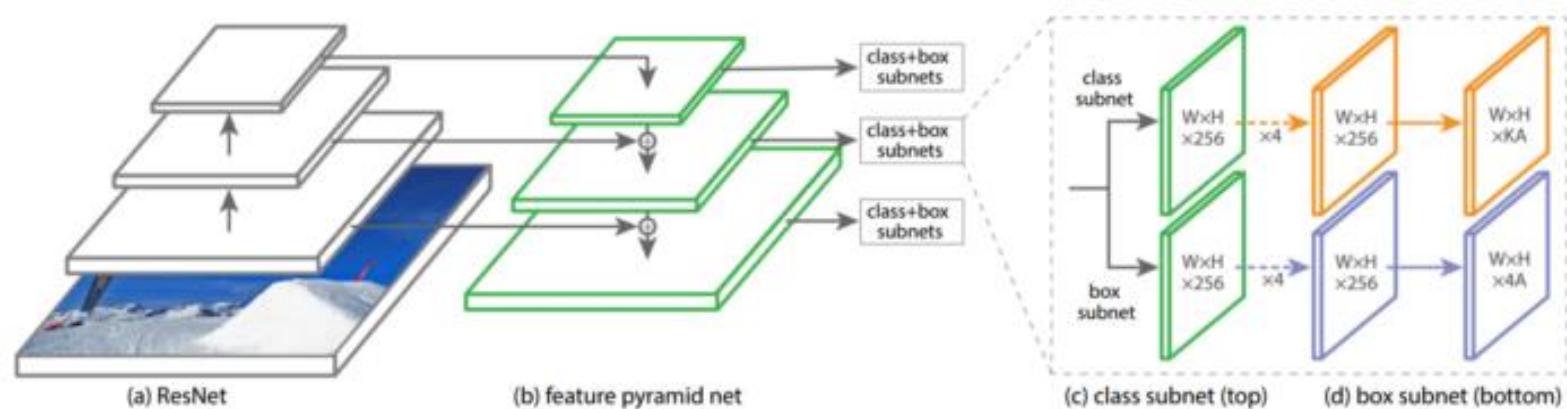
## RetinaNet

RetinaNet is more about proposing a new loss function to handle class imbalance, rather than publishing a novel new network. Let's bring the class imbalance back again. One-stage detectors such as SSD and YOLO are apparently faster but still fall behind in accuracy compared to two-stage detectors. And the **class imbalance** issue is one of the reasons for this drawback.



So the authors proposed a new loss function called **focal loss** by putting a weight on easy examples. The mathematical expression is as shown above. It's a cross-entropy loss multiplied with a modulating factor. The modulating factor reduces the impact of easy examples on the loss.

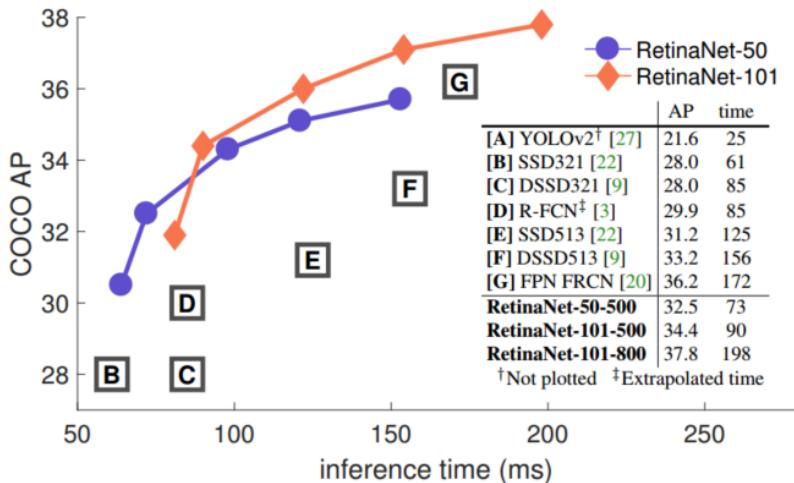
For example, compare the loss when  $P_t=0.9$  and  $\gamma=2$ . If we say the cross-entropy loss as  $CE$ , then the focal loss becomes  $-0.01CE$ . The loss becomes 100 times lower. And if the  $P_t$  gets bigger, say 0.968, with the constant  $\gamma$ , the focal loss becomes  $-0.001CE$  (as  $(1-0.968)^2 = (0.032)^2 \approx 0.001$ ). So it puts down the easy examples even harder and adjusting the imbalance in return. And when  $\gamma$  increases, as you can see the graph on the right, the loss gets smaller as the weight increases when  $P_t$  is constant.



The architecture of RetinaNet is as shown above. As we now know ResNet, RPN of Faster R-CNN and FPN, there is nothing new here. The network can be divided into two parts, a backbone with (a) and (b), and two subnets for classification and box regression. The backbone network is composed of ResNet and FPN and the pyramid has levels P3 to P7.

As with RPN, there are prior anchor boxes as well. The size of anchors changes according to its level. At the level P3, the anchor area is 32\*32 and at P7, it's 512\*512. The anchors have three different ratios and three different sizes, so the number of anchors A for each level is 9.

Therefore, the dimension of output from a box regression subnet is  $(N \times N) \times 4A$ . And when the number of classes in the dataset is K, the output from a classification subnet becomes  $(N \times N) \times KA$ .



The result was quite eye-catching. RetinaNet outperformed all the previous networks in both accuracy and speed. RetinaNet-101-500 indicates the network with ResNet-101 and a 500-pixel image scale. Using larger scales gives higher accuracy than all two-stage approaches, while still being fast enough as well.

## FOCAL LOSS

**Loss functions** are mathematical equations that calculate how far the predictions deviate from the actual values. Higher loss values suggest that the model is making a significant error, whereas lower loss values imply that the predictions are rather accurate. The goal is to reduce the loss function as much as possible. The loss function is used by models to learn the trainable parameters, such as weights and biases. Because the weight updation equation of the parameters has the first derivative of the loss function with respect to the weights or biases, the behavior of this function will have a significant impact on the gradient descent process.

$$w_{t+1} = w_t - \alpha \frac{\partial C}{\partial w_t}$$

There are numerous loss functions available right now. Each of them has a different mathematical equation and a different method of penalizing the model's errors. There are also benefits and drawbacks to each, which we must weigh before deciding on the best function to use.

Now that we've defined the loss function, let's go over the issues that Categorical Cross-Entropy loss causes and how Focal loss solves them.

*Categorical Cross-Entropy Loss*

**Categorical Cross-Entropy loss** is traditionally used in classification tasks. As the name implies, the basis of this is Entropy. In statistics, entropy refers to the disorder of the system. It quantifies the degree of uncertainty in the model's predicted value for the variable. **The sum of the entropies of all the probability estimates is the cross entropy.**

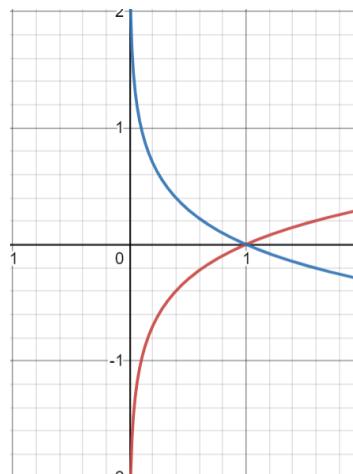
$$\text{Entropy} = -p_i \log_b(p_i)$$

$$\text{Cross Entropy} = CE = - \sum_{i=1}^n Y_i \log_b(p_i)$$

Where  $Y$  is the **true label** and  $p$  is the **predicted probability**.

**Note:** The formula shown above is for a discrete variable. In the case of a continuous variable, the summation should be replaced by integration.

The logarithm graph clearly shows that the summation will be negative because probabilities range from 0 to 1. As a result, we add a minus to invert the sign of the summation term. The graph of  $\log(x)$  and  $-\log(x)$  is shown below (x).



Cases where Cross-Entropy loss performs badly

- **Class imbalance** inherits bias in the process. The majority class examples will dominate the loss function and gradient descent, causing the weights to update in the direction of the model becoming more confident in predicting the majority class while putting less emphasis on the minority classes. **Balanced Cross-Entropy** loss handles this problem.
- Fails to distinguish between hard and easy examples. Hard examples are those in which the model repeatedly makes huge errors, whereas easy examples are those which are easily classified. As a result, Cross-Entropy loss fails to pay more attention to hard examples.

*Balanced Cross-Entropy Loss*

Balanced Cross-Entropy loss adds a weighting factor to each class, which is represented by the letter  $\alpha$ ,  $[0, 1]$ . Alpha could be the inverse class frequency or a hyper-parameter that is

determined by cross-validation. The alpha parameter replaces the actual label term in the Cross-Entropy equation.

$$\text{Balanced Cross Entropy} = - \sum_{i=1}^n \alpha_i \log_b(p_i)$$

Despite the fact that this loss function addresses the issue of class imbalance, it cannot distinguish between hard and easy examples. The problem was solved by focal loss.

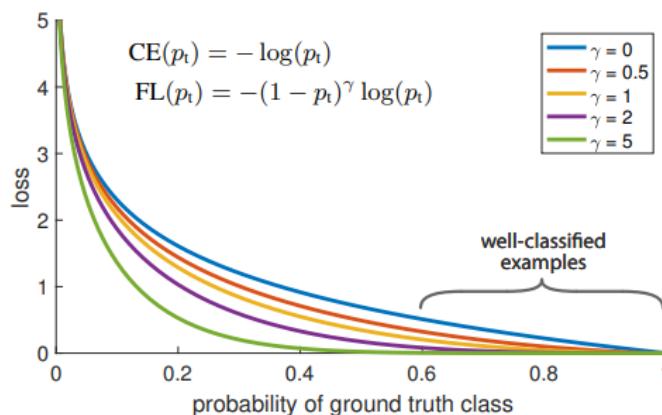
### Focal Loss

Focal loss focuses on the examples that the model gets wrong rather than the ones that it can confidently predict, ensuring that **predictions on hard examples improve over time rather than becoming overly confident with easy ones.**

How exactly is this done? Focal loss achieves this through something called **Down Weighting**. Down weighting is a technique that reduces the influence of easy examples on the loss function, resulting in more attention being paid to hard examples. This technique can be **implemented by adding a modulating factor to the Cross-Entropy loss.**

$$\text{Focal Loss} = \text{FE} = - \sum_{i=1}^n (1 - p_i)^\gamma \log_b(p_i)$$

Where  $\gamma$  is the **focusing parameter** to be tuned using cross-validation. The image below shows how Focal Loss behaves for different values of  $\gamma$ .



How Gamma parameter works?

- In the case of the misclassified sample, the  $p_i$  is small, making the modulating factor approximately or very close to 1. That keeps the loss function unaffected. As a result, it behaves as a Cross-Entropy loss.
- As the confidence of the model increases, that is,  $p_i \rightarrow 1$ , modulating factor will tend to 0, thus down-weighting the loss value for well-classified examples. The focusing parameter,  $\gamma \geq 1$ , will rescale the modulating factor such that the easy examples are down-weighted more than the hard ones, reducing their impact on the loss function. For

instance, consider predicted probabilities to be 0.9 and 0.6. Considering  $\gamma = 2$ , the loss value calculated for 0.9 comes out to be 4.5e-4 and down-weighted by a factor of 100, for 0.6 to be 3.5e-2 down-weighted by a factor of 6.25. From the experiments,  $\gamma = 2$  worked the best for the authors of the Focal Loss paper.

- When  $\gamma = 0$ , Focal Loss is equivalent to Cross Entropy.

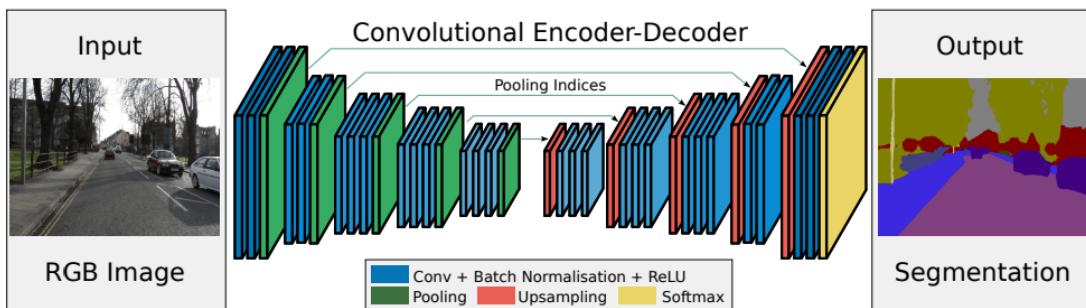
In practice, we use an  **$\alpha$ -balanced variant** of the focal loss that inherits the characteristics of both the **weighing factor  $\alpha$**  and the **focusing parameter  $\gamma$** , yielding slightly better accuracy than the non-balanced form.

$$\text{Focal Loss} = \text{FE} = - \sum_{i=1}^n \alpha_i (i - p_i)^\gamma \log_b(p_i)$$

Focal Loss naturally solved the problem of class imbalance because examples from the majority class are usually easy to predict while those from the minority class are hard due to a lack of data or examples from the majority class dominating the loss and gradient process. Because of this resemblance, the Focal Loss may be able to solve both problems.

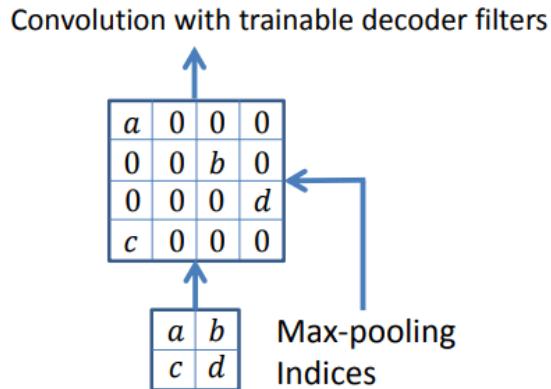
## SegNet

One tool for **semantic segmentation** is [SegNet](#). It's an **encoder-decoder network**:



During the **encoder** phase, it downsamples the input image with max-pooling layers after the convolution through 13 layers. After each pooling step, it saves the indices from each filter together with the maximum values in each subgrid. The indices indicate where the pixels originally were.

Then, the decoder stage starts. After applying max-pooling filters during encoding, we get low-resolution **feature** maps that need to be upsampled. To perform this upsample, SegNet uses the indices it saved during encoding. It restores the pixels that defined the max-pooling and fills the rest with zeros:

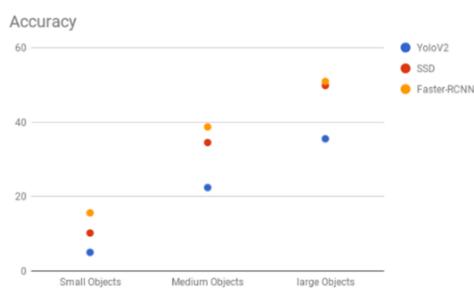


Lastly, this sparse map, made of zeros and the max-pooling values, is convolved to provide us with dense feature maps.

## Which algorithm to use?

Choice of a right object detection method is crucial and depends on the problem you are trying to solve and the setup. Object Detection is the backbone of many practical applications of computer vision such as autonomous cars, security, surveillance, and many industrial applications.

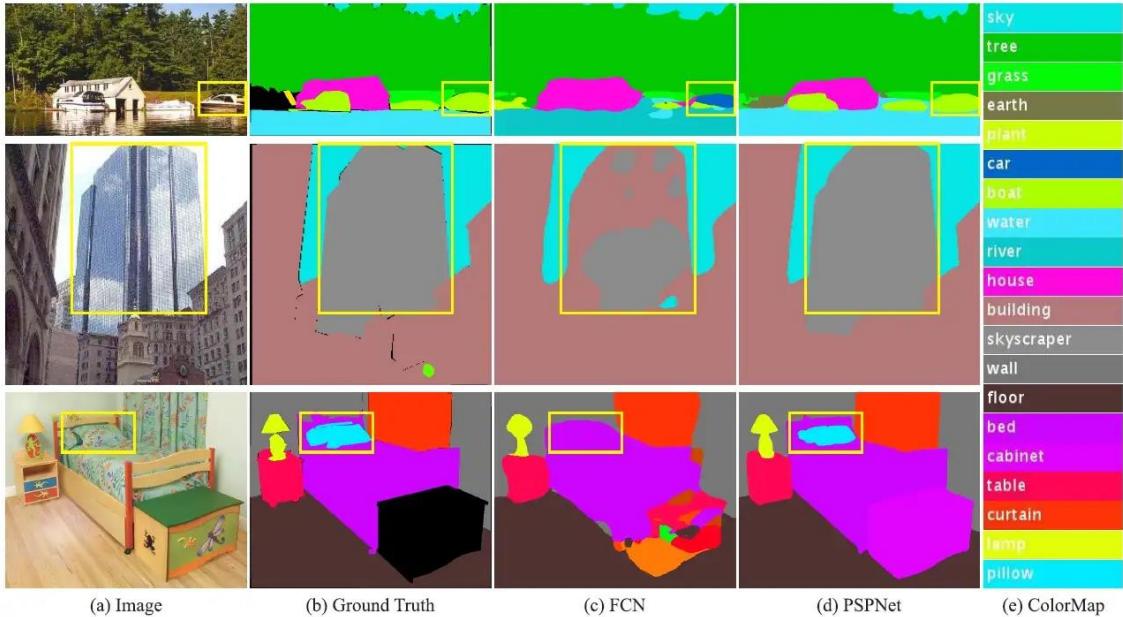
- **Faster-RCNN** is the choice if you mostly care about the accuracy numbers.
- **SSD** provides good balance between performance and speed.
- **YOLO** is super fast but on the account of performance.



## Pyramid Scene Parsing Network (PSPNet)

State-of-the-art scene parsing frameworks are mostly based on the fully convolutional network (FCN). The deep convolutional neural network (CNN) based methods boost dynamic object

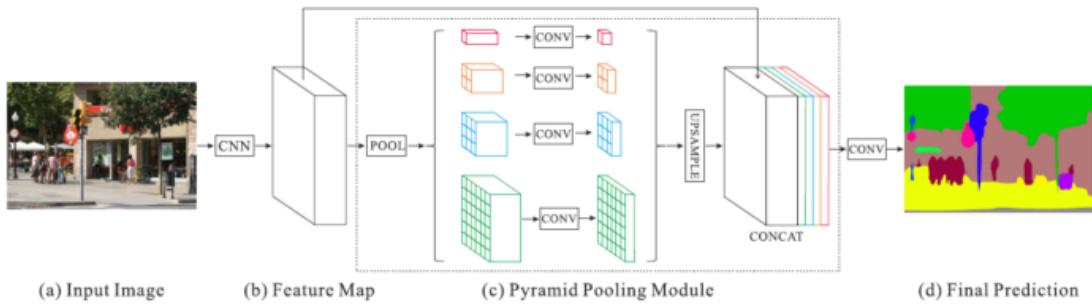
understanding, and yet still face challenges considering diverse scenes and unrestricted vocabulary. An example is where a boat is mistaken as a car. These errors are due to similar appearance of objects. But when viewing the image regarding the context prior that the scene is described as boathouse near a river, correct prediction should be yielded.



From the image above we can see **the need of Global Information**:

- **Mismatched Relationship:** FCN predicts the boat in the yellow box as a “car” based on its appearance. But the common knowledge is that a car is seldom over a river.
- **Confusion Categories:** FCN predicts the object in the box as part of skyscraper and part of building. These results should be excluded so that the whole object is either skyscraper or building, but not both.
- **Inconspicuous Classes:** The pillow has similar appearance with the sheet. Overlooking the global scene category may fail to parse the pillow.

## PYRAMID POOLING METHOD



(a) and (b)

At (a), we have an input image. At (b), ResNet is used with dilated network strategy (DeepLab / DilatedNet) for extracting features. The dilated convolution is following DeepLab. The feature map size is 1/8 of the input image here.

(c)

Sub-Region Average Pooling

At (c), **sub-region average pooling is performed for each feature map.**

- **Red:** This is the **coarsest level** which perform global average pooling over each feature map, to **generate a single bin output**.
- **Orange:** This is the **second level** which divide the feature map into  **$2 \times 2$  sub-regions**, then perform **average pooling for each sub-region**.
- **Blue:** This is the **third level** which divide the feature map into  **$3 \times 3$  sub-regions**, then perform **average pooling for each sub-region**.
- **Green:** This is the **finest level** which divide the feature map into  **$6 \times 6$  sub-regions**, then perform **pooling for each sub-region**.

1x1 Convolution for Dimension Reduction

Then **1x1 convolution** is performed **for each pooled feature map to reduce the context representation to  $1/N$  of the original one** (black) if the level size of pyramid is N.

- In this example, N=4 because there are 4 levels in total (red, orange, blue and green).
- If the number of input feature maps is 2048, then the output feature map will be  $(1/4) \times 2048 = 512$ , i.e. 512 number of output feature maps.

Bilinear Interpolation for Upsampling

Bilinear interpolation is performed to up-sample each low-dimension feature map to have the same size as the original feature map (black).

Concatenation for Context Aggregation

All different levels of upsampled feature maps are concatenated with the original feature map (black). These feature maps are fused as global prior. That is the end of pyramid pooling module at (c).

(d)

Finally, it is followed by a convolution layer to generate the final prediction map at (d).

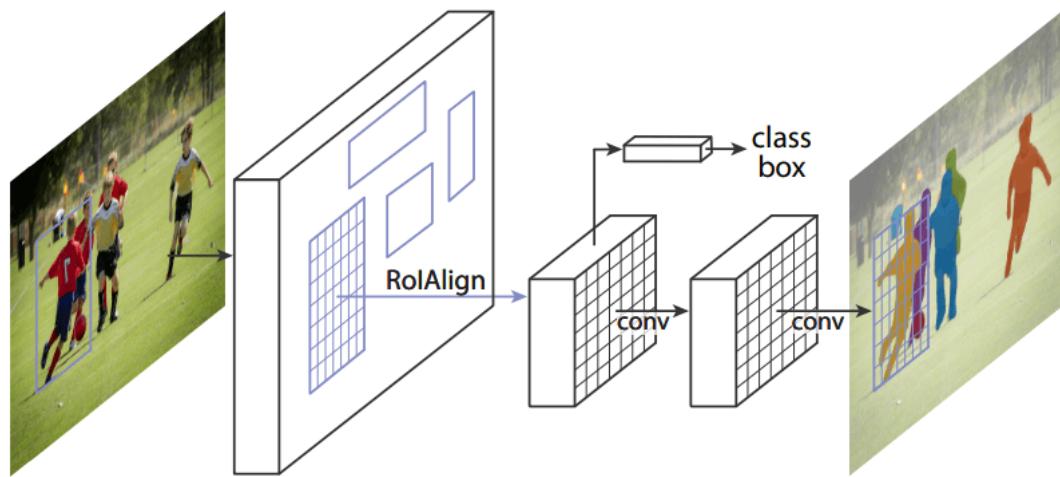
The idea of sub-region average pooling actually is quite similar to **Spatial Pyramid Pooling** in SPPNet. The  $1 \times 1$  convolution then concatenation is quite similar to the depthwise convolution in Depthwise Separable Convolution used by Xception or MobileNetV1 as well, except that bilinear interpolation is used to make all the feature maps' sizes equal.

## Mask R-CNN

One of the most eminent methods for **instance segmentation** is Mask R-CNN.

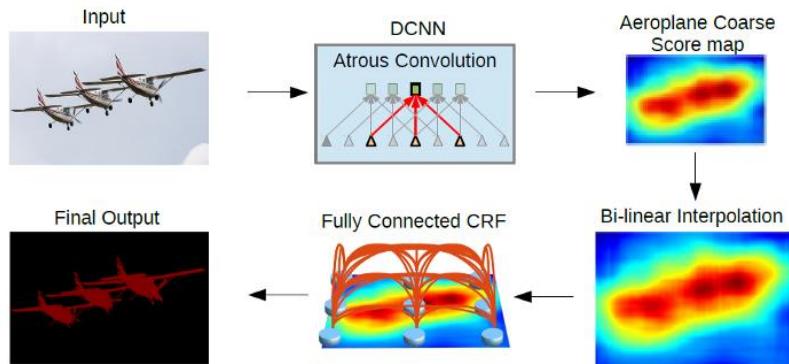
Mask R-CNN works in stages. The first stage is the Region Proposal Network (RPN), which defines the candidates for bounding.

In the next step, the class is predicted along with the bounding box offset and a mask for each Region of Interest. The main novelty of this method is the parallel bounding-box classification and regression:



# DeepLab

## DeepLab V2



The above figure is the **DeepLab model architecture**. First, the **input image goes through the network** with the use of **atrous convolution** and **ASPP**. Then the **output from the network** is **bilinearly interpolated** and goes through the **fully connected CRF** to **fine tune the result** and get the **final output**.

### Atrous Convolution

The term “**Atrous**” indeed comes from French “à trous” meaning hole. Thus, it is also called “algorithme à trous” and “**hole algorithm**”. Some of the papers also call this “**dilated convolution**”. It is commonly used in wavelet transform and right now it is applied in convolutions for deep learning.

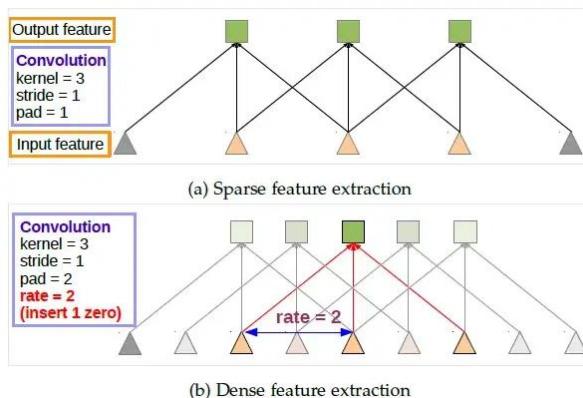
Below is the equation of **atrous convolution**:

$$y[i] = \sum_{k=1}^K x[i + r \cdot k]w[k]$$

When:

- $r = 1$ , it is the standard convolution we usually use.
- $r > 1$ , it is the **atrous convolution** which is the **stride to sample the input** sample during convolution.

The below figure illustrate the idea:



The idea of atrous convolution is simple. At the top of the figure above, it is the standard convolution.

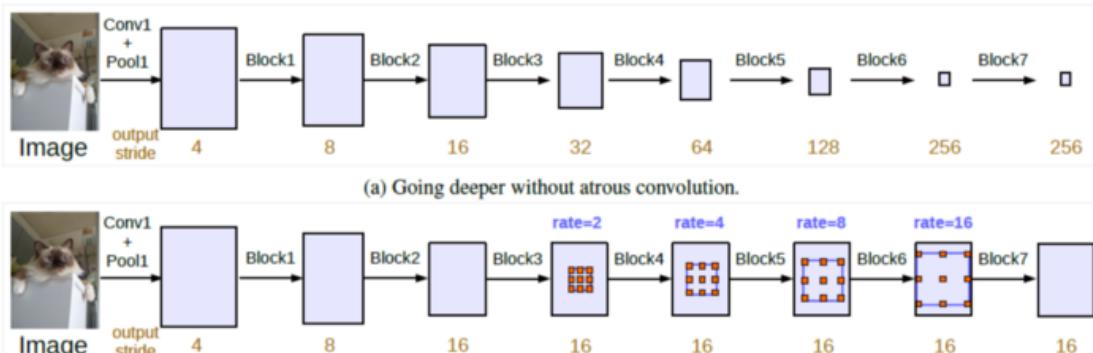
At the bottom of the figure, it is the **atrous convolution**. We can see that when rate = 2, the input signal is sampled alternatively. First, pad=2 means we pad 2 zeros at both left and right sides. Then, with rate=2, we sample the input signal every 2 inputs for convolution. Thus, at the output, we will have 5 outputs which **makes the output feature map larger**.

If we remember FCN, a series of convolution and pooling makes the output feature map very small, and need  $32 \times$  upsampling which is an aggressive upsampling.

Also, **atrous convolution allows us to enlarge the field of view of filters to incorporate larger context**. It thus offers an efficient mechanism to control the field-of-view and finds the best trade-off between accurate localization (small field-of-view) and context assimilation (large field-of-view).

In DeepLab, using VGG-16 or ResNet-101, the stride of last pooling (pool5) or convolution conv5\_1 is set to 1 respectively to avoid signal from decimated too much. And atrous convolution is used to replace all subsequent convolutional layers using rate = 2. **The output is much larger**. We only need to have  $8 \times$  upsampling to upsample the output. And **bilinear interpolation** has quite good performance for  $8 \times$  upsampling.

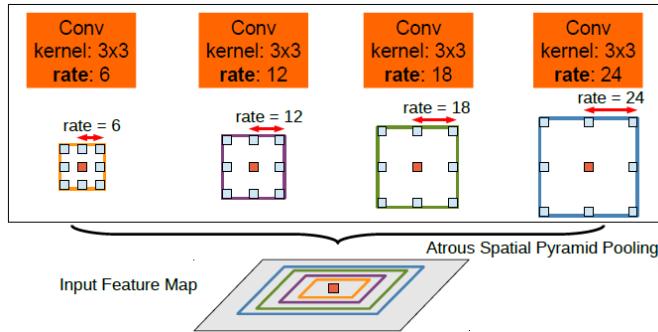
#### *Going Deeper with Atrous Convolution Using Multi-Grid*



(b) Going deeper with atrous convolution. Atrous convolution with  $rate > 1$  is applied after block3 when  $output\_stride = 16$ .

- **(a) Without Atrous Conv:** Standard conv and pooling are performed which makes the output stride increasing, i.e. the output feature map smaller, when going deeper. However, **consecutive striding is harmful for semantic segmentation because location/spatial information is lost at the deeper layers**.
- **(b) With Atrous Conv:** With atrous conv, we can keep the stride constant but with **larger field-of-view without increasing the number of parameters** or the amount of computation. And finally, we can have **larger output feature map which is good for semantic segmentation**.

## Atrous Spatial Pyramid Pooling (ASPP)



ASPP actually is an atrous version of SPP, in which the concept has been used in SPPNet. In ASPP, parallel atrous convolution with different rate applied in the input feature map, and fuse together.

As objects of the same class can have different scales in the image, ASPP **helps to account for different object scales** which can improve the accuracy.

## Fully Connected Conditional Random Field (CRF)

Fully Connected CRF is applied at the network output after bilinear interpolation:

$$E[x] = \sum_i \theta_i(x_i) + \sum_{ij} \theta_{ij}(x_i, x_j)$$

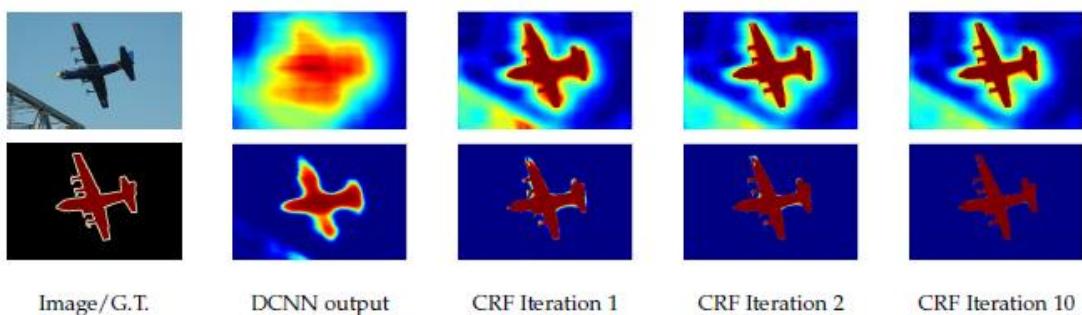
where:

$$\theta_i(x_i) = -\log P(x_i)$$

$$\theta_{ij}(x_i, x_j) = \mu(x_i, x_j) \left[ w_1 \exp \left( -\frac{\|p_i - p_j\|^2}{2\sigma_\alpha^2} - \frac{\|I_i - I_j\|^2}{2\sigma_\beta^2} \right) + w_2 \exp \left( -\frac{\|p_i - p_j\|^2}{2\sigma_\gamma^2} \right) \right]$$

$x$  is the label assignment for pixels.  $P(x_i)$  is the label assignment probability at pixel  $i$ . Therefore the first term  $\theta_i$  is the log probability.

For the second term,  $\theta_{ij}$ , it is a filter.  $\mu = 1$  when  $x_i \neq x_j$ .  $\mu = 0$  when  $x_i = x_j$ . In the bracket, it is the weighted sum of two kernels. The **first kernel** depends on pixel value difference and pixel position difference, which is a **kind of bilateral filter**. **Bilateral filter has the property of preserving edges**. The **second kernel** only **depends on pixel position difference**, which is a **Gaussian filter**. Those  $\sigma$  and  $w$ , are found by cross validation. The number of iteration is 10.

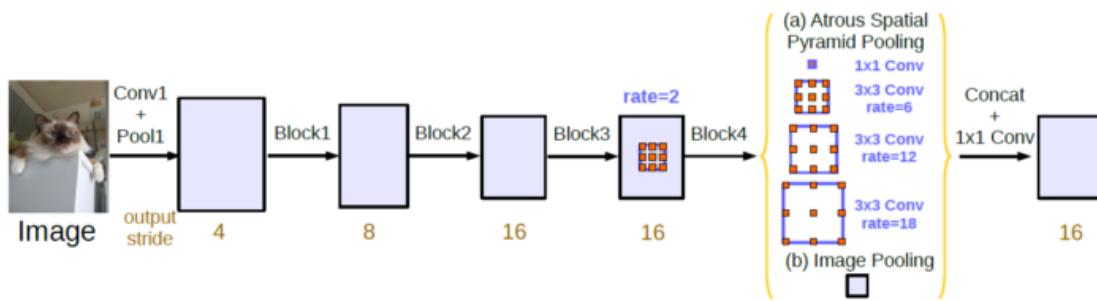


With 10 times of CRF, those small areas with different colors around the aero plane are smoothed out successfully.

However, CRF is a post-processing process which makes DeepLabv1 and DeepLabv2 become not an end-to-end learning framework. And it is **NOT used in DeepLabv3 and DeepLabv3+ already**.

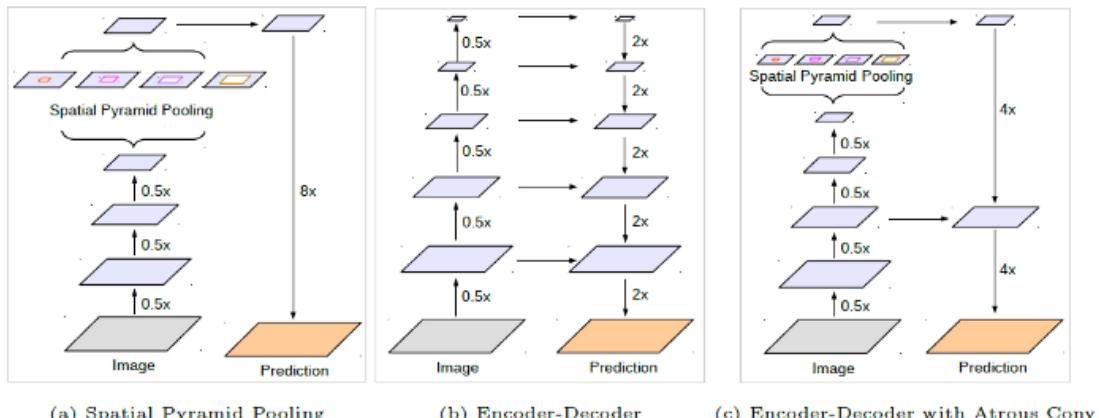
### DeepLab V3

Authors tried to REthink or restructure the DeepLab architecture and finally come up with a more enhanced DeepLabv3. DeepLabv3 outperforms DeepLabv1 and DeepLabv2, even with the **post-processing step Conditional Random Field (CRF) removed**.



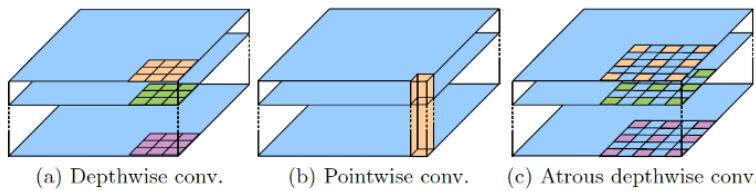
The resulting features from all the branches are then **concatenated** and pass through another **1x1 convolution** (also with 256 filters and batch normalization) before the final 1x1 convolution which generates the final logits.

### DeepLab V3+



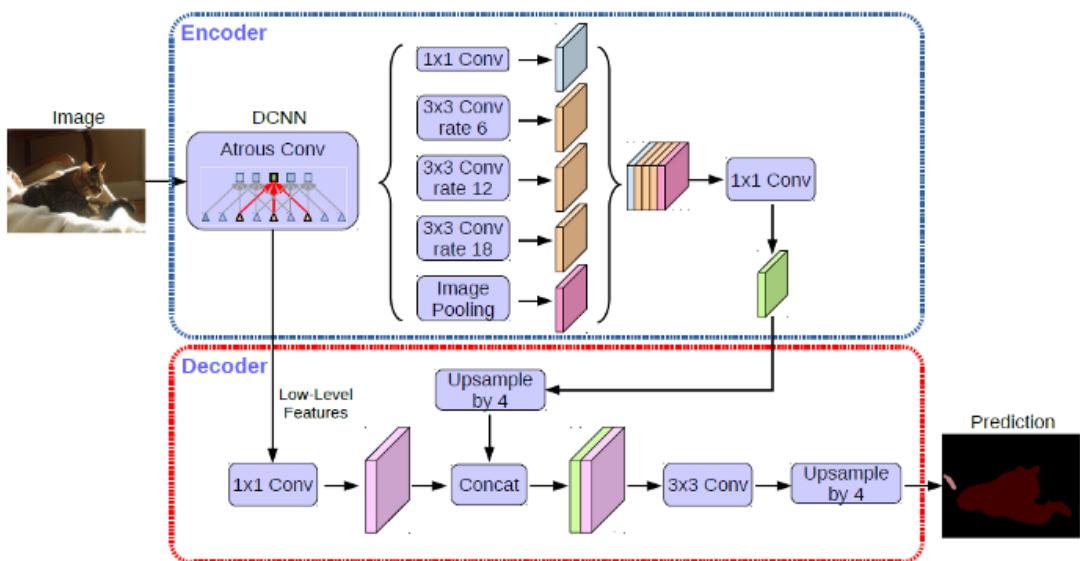
- (a) With **Atrous Spatial Pyramid Pooling (ASPP)**, able to encode multi-scale contextual information.
- (b) With **Encoder-Decoder Architecture**, the location/spatial information is recovered.
- (c) DeepLabv3+ makes use of (a) and (b).

## Atrous Separable Convolution



- (a) and (b) **Depthwise Separable Convolution**: It factorizes a standard convolution into a **depthwise convolution followed by a point-wise convolution** (i.e.,  **$1 \times 1$  convolution**), **drastically reduces computation complexity**.
- (c) **Atrous Depthwise Convolution**: Atrous convolution is supported in the depthwise convolution. And it is found that it significantly reduces the computation complexity of proposed model while maintaining similar (or better) performance.

## Encoder-Decoder Architecture

*DeepLab V3 as Encoder*

- For the task of **image classification**, the spatial resolution of the final feature maps is usually 32 times smaller than the input image resolution and thus output stride = 32.
- For the task of **semantic segmentation**, **it is too small**.
- One can adopt output stride = 16 (or 8) for denser feature extraction by removing the striding in the last one (or two) block(s) and **applying the atrous convolution** correspondingly.
- Additionally, DeepLabv3 augments the Atrous Spatial Pyramid Pooling module, which probes convolutional features at multiple scales by applying atrous convolution with different rates, with the image-level features.

*Proposed Decoder*

- The encoder features are **first bilinearly upsampled by a factor of 4 and then concatenated with the corresponding low-level features.**
- There is **1x1 convolution on the low-level features** before concatenation to reduce the number of channels, since the corresponding low-level features usually contain a large number of channels (e.g., 256 or 512) which may outweigh the importance of the rich encoder features.
- After the concatenation, we apply a few 3x3 convolutions to **refine the features followed by another simple bilinear upsampling by a factor of 4.**
- This is much better comparing the one bilinearly upsampling 16x directly.

These following models take entire images as input and use a deep neural network to directly output **disparity maps**.

## DispNet

**DispNet** is the first end-to-end trained deep neural network for stereo. It takes a pair of left and right images as input and has a **U-Net** like architecture. This includes convolutions, downsampling and skip-connections to retain details when increasing the resolution again. Note that there is no explicit global optimization.

### SPECIFIC FOR THIS ARCHITECTURE

#### *Correlation Layer*

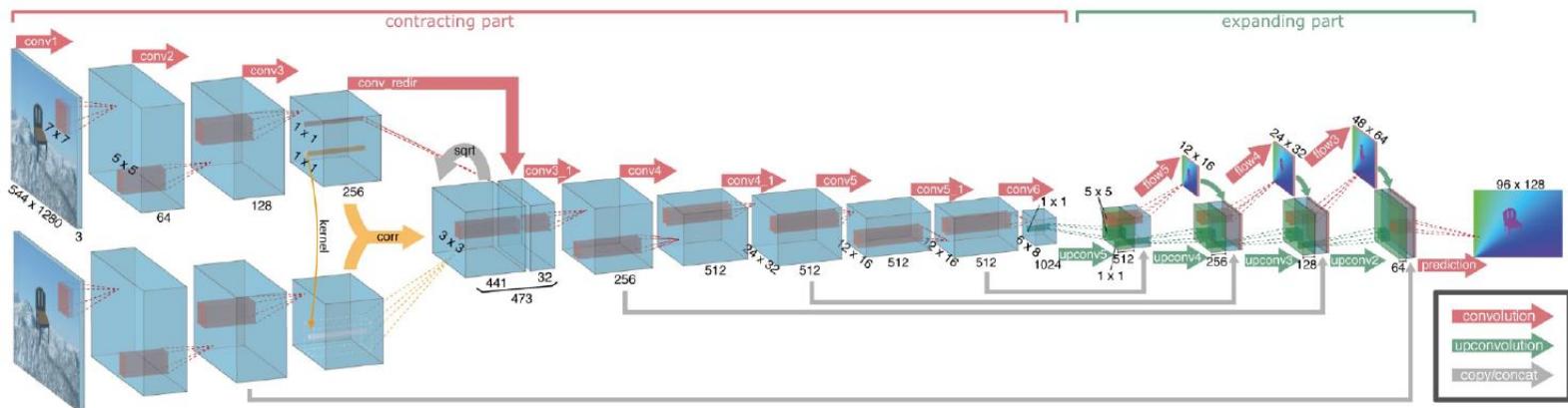
After a few convolutions and pooling layers, a correlation layer (40px displacement corresponding to 160px in input image) is applied to the feature outputs. This layer works similar to the correlation during block matching.

#### *Multi-Scale Loss*

During training, a multi-scale loss (disparity errors in pixels) is applied with downscaled versions of the ground truth at intermediate layers.

#### *Curriculum Learning*

First, the model is trained on simple scenes and small resolutions. Over time, the difficulty is increased until the target dataset is reached.

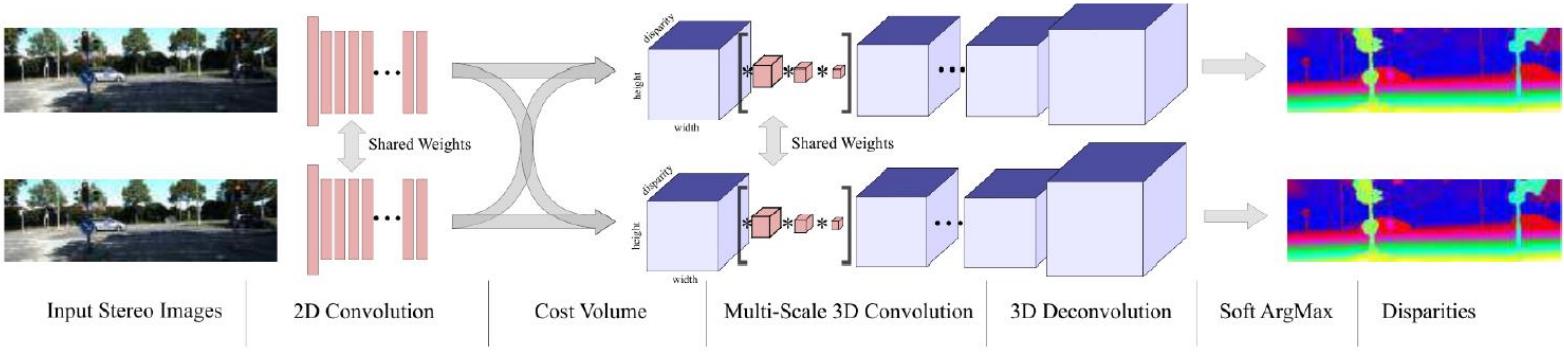


## GC-Net

The **GC-Net** is a follow-up work of DispNet and improves performance even further. The key idea is calculate a disparity cost volume and apply 3D convolutions on it instead of 2D convolutions. It learns a matching cost  $c_\Theta(d)$  which can be converted to a disparity via the expectation:

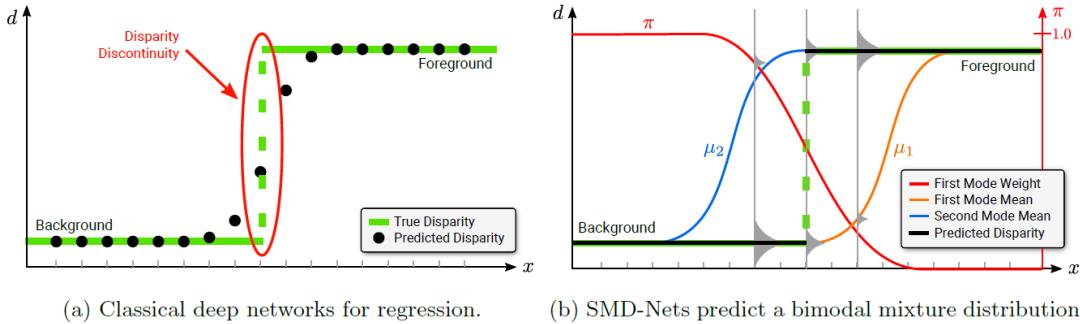
$$d^* = E[d] = \sum_{d=0}^D \frac{\text{softmax}(-c_\Theta(d))}{p(d)} d$$

Although this model has slightly better performance, the memory requirements are larger as well (due to the 3D feature volume).

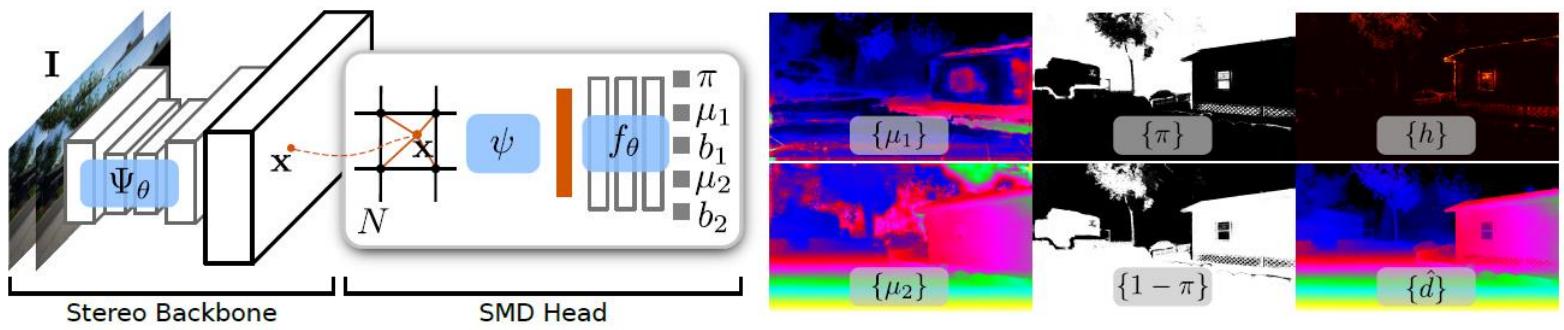


## Stereo Mixture Density Networks (SMD-Net)

In contrast to other architectures, SMD-Nets predict sharper boundaries at higher resolutions. Classical deep networks for stereo regression suffer from a smoothness bias and hence continuously interpolate object boundaries. Instead, SMD-Nets predict a bimodal (Laplacian) mixture distribution which allows to accurately capture uncertainty close to depth discontinuities.



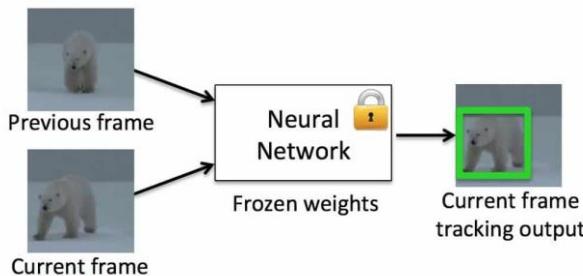
Regarding the architecture, SMD-Nets use a 2D or 3D convolutional backbone in combination with a shallow MLP head that regresses the distribution parameters from interpolated features. This enables training and inference at arbitrary spatial resolutions.



The following are **deep object tracking algorithms**.

## Generic Object Tracking Using Regression Networks (GOTURN)

**Generic Object Tracking Using Regression Networks (GOTURN)** is a **Deep Learning based** tracking algorithm. While most tracking algorithms are trained in an online manner (the algorithm learns the appearance of the object it is tracking at runtime) and are typically much faster than a Deep Learning based solution. GOTURN is tracking by learning the motion of an object in an offline manner. The **GOTURN model is trained on thousands of video sequences and does not need to perform any learning at runtime.**



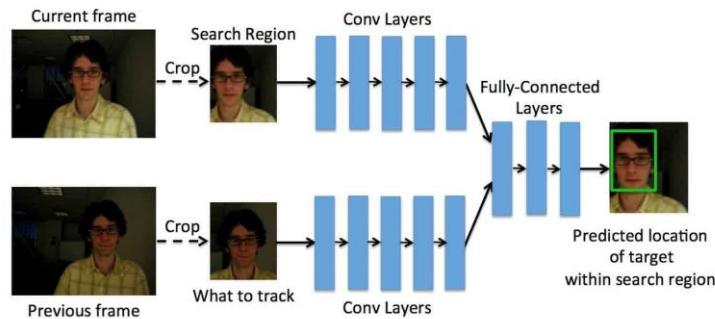
**GOTURN is trained using a pair of cropped frames** from thousands of videos.

In the **first frame** (also referred to as the previous frame), the **location of the object is known**, and **the frame is cropped to two times the size of the bounding box around the object**. The object in the first cropped frame is **always centered**.

The location of the object in the **second frame** (also referred to as the current frame) **needs to be predicted**. The **bounding box used to crop the first frame is also used to crop the second frame**. Because the object might have moved, the **object is not centered in the second frame**.

A **Convolutional Neural Network (CNN)** is trained to predict the **location of the bounding box** in the second frame.

## ARCHITECTURE



GOTURN takes two cropped frames as input. Notice, the previous frame, shown at the bottom, is centered and our goal is the find the bounding box for the current frame shown on the top.

Both frames pass through a bank of convolutional layers. The layers are simply the first five convolutional layers of the CaffeNet architecture. The outputs of these convolutional layers (i.e. the pool5 features) are concatenated into a single vector of length 4096. This vector is input to 3 fully connected layers. The last fully connected layer is finally connected to the output layer containing 4 nodes representing the top and bottom points of the bounding box.

**GOTURN can run very fast i.e. 100fps** on a GPU powered machine.

# Multi-Domain Convolutional Neural Network Tracker (MDNet)

Here is the algorithm described by the authors in the paper that explains the online tracking:

---

**Algorithm 1** Online tracking algorithm

---

```

Input : Pretrained CNN filters  $\{w_1, \dots, w_5\}$ 
        Initial target state  $x_1$ 
Output: Estimated target states  $x_t^*$ 
1: Randomly initialize the last layer  $w_6$ .
2: Train a bounding box regression model.
3: Draw positive samples  $S_1^+$  and negative samples  $S_1^-$ .
4: Update  $\{w_4, w_5, w_6\}$  using  $S_1^+$  and  $S_1^-$ ;
5:  $\mathcal{T}_s \leftarrow \{1\}$  and  $\mathcal{T}_l \leftarrow \{1\}$ .
6: repeat
7:   Draw target candidate samples  $x_t^i$ .
8:   Find the optimal target state  $x_t^*$  by Eq. (1).
9:   if  $f^+(x_t^*) > 0.5$  then
10:    Draw training samples  $S_t^+$  and  $S_t^-$ .
11:     $\mathcal{T}_s \leftarrow \mathcal{T}_s \cup \{t\}$ ,  $\mathcal{T}_l \leftarrow \mathcal{T}_l \cup \{t\}$ .
12:    if  $|\mathcal{T}_s| > \tau_s$  then  $\mathcal{T}_s \leftarrow \mathcal{T}_s \setminus \{\min_{v \in \mathcal{T}_s} v\}$ .
13:    if  $|\mathcal{T}_l| > \tau_l$  then  $\mathcal{T}_l \leftarrow \mathcal{T}_l \setminus \{\min_{v \in \mathcal{T}_l} v\}$ .
14:    Adjust  $x_t^*$  using bounding box regression.
15:   if  $f^+(x_t^*) < 0.5$  then
16:     Update  $\{w_4, w_5, w_6\}$  using  $S_{v \in \mathcal{T}_s}^+$  and  $S_{v \in \mathcal{T}_s}^-$ .
17:   else if  $t \bmod 10 = 0$  then
18:     Update  $\{w_4, w_5, w_6\}$  using  $S_{v \in \mathcal{T}_l}^+$  and  $S_{v \in \mathcal{T}_l}^-$ .
19: until end of sequence

```

---

First, we **randomly initialize a new fully connected layer** which will be coupled with our shared layers. This layer will be **responsible for learning the target features** particular to our new domain.

The **initial coordinates** of the object we want to track will be **known to us**. We also **initialize two arrays** which will **store feature maps obtained from passing the positive and negative samples** through our model as we will see later.

Similar to the offline pre-training, a set of **positive and negative boxes are sampled around the target box of our first frame**. These boxes are then **warped** from the images and passed through our pre-trained network. We store the features obtained from the conv3 layer into the 2 arrays mentioned previously.

**Note :** Unlike the pre-training phase, a much larger number of negative boxes are sampled in this stage and only those boxes which give a high positive score (incorrect prediction) are saved to the array (**hard negative mining**).

The conv3 features in our arrays are now used to train the fc4-fc6 layers.

In brief, we have trained our model using a set of positive samples and a set of negative samples obtained through hard negative mining, from the first image of our new test sequence. This **helps our model learn the features of the new target** we will be tracking.

Taking inspiration from the modern object detectors, the authors also add a **Ridge regressor** to **refine the location of our positive samples**. This regressor is trained only once (as it will be time-consuming to train it on every image) on the samples obtained from the first image and will be used to fine-tune the location of positive samples in subsequent images (step 2).

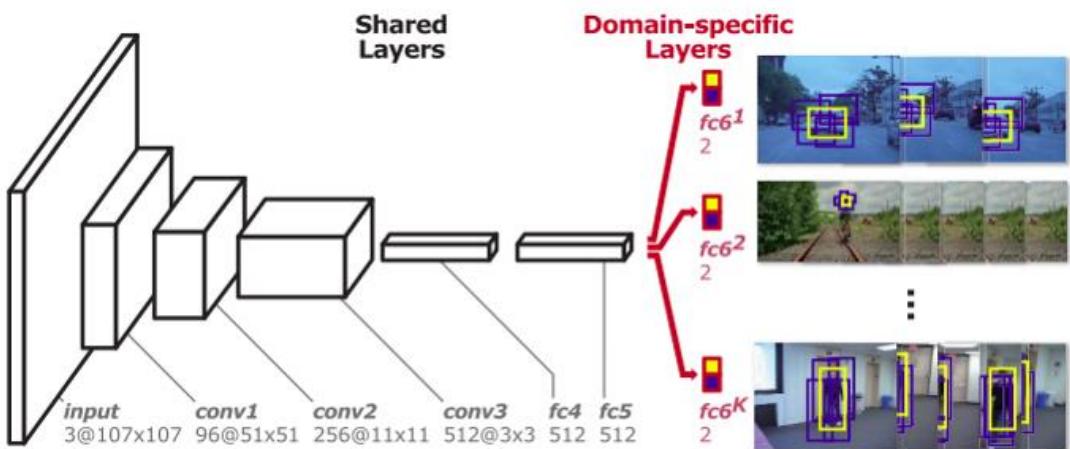
From the second frame onwards, we do the following:

- Randomly sample around the previous location of our target on the new image.
- Pass these samples through our network and take the **top 5 positive boxes**.
- The mean of the coordinates of these 5 boxes will be our predicted box and the mean score will be our predicted score.
- If the **predicted score > 0.5**, we have a **successful prediction** and we **refine the location using our regressor**. To enable continual learning, we again sample around this new predicted box to get a set of positive and negative boxes. The features obtained from the conv3 layer, after passing these boxes into our network are added to our two arrays.
- If our **predicted score < 0.5**, we **keep the prediction of our previous frame**. We then update the fc4-fc6 layers using the positive and negative features that we have saved so far in our arrays.
- After every 10 iterations, we again update the fc4-fc6 layers using the features that we saved in the arrays.

We impose a limit on the number of conv3 features we save in the arrays, deleting old features in case the size crosses this limit. (steps 12 and 13)

In this way, we proceed to make predictions for all our images in the sequence.

## ARCHITECTURE

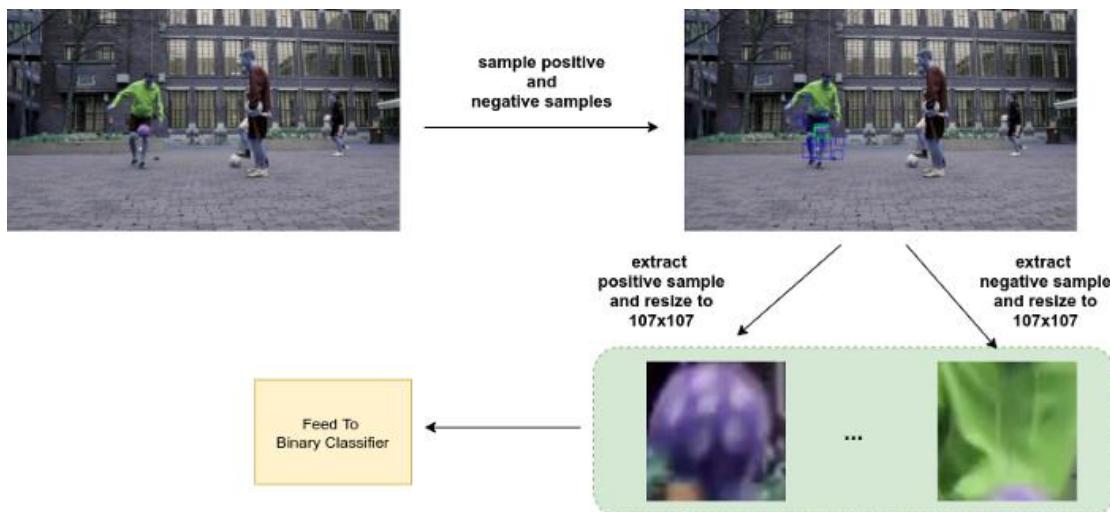


We can break down MDNet into two parts. The **first part** consisting of **conv1-fc5 layers** is **responsible for learning domain independent features** and is shared among all the video sequences. The **fc6 layer** is **unique to each sequence** and is **responsible for learning the domain specific features**. So if we have K video sequences, we would be having K fully connected layers, one for each sequence. This means that during training, the gradients would be backpropagated

among the shared layers irrespective of the sequence, but only for fc6(i) among the fc6 layers, where  $i$  is the sequence number of the current image being processed. **The output of the network is a score indicating whether the input image contains the target object or is background.**

## PRE-TRAINING

Before testing our model on new video sequences, we must train our model. Say our dataset contains  $K$  video sequences, each sequence containing a certain number of images with our target object annotated. We loop through all video sequences in one epoch. The authors take one minibatch consisting of 8 images, and from each image we uniformly sample 50 positive and 200 negative locations around the target object. A **positive sample has an IoU (intersection over union)** with the **target object greater than 0.7**, whereas a **negative sample has an IoU less than 0.5** with the target object. Again, these are just hyperparameters which can be adjusted to suit our dataset. These regions are then warped from the image and resized to 107x107 px and then fed into our binary classifier described above, to learn the domain-agnostic and domain-specific features. The figure below illustrates this more precisely.



In this way, we train our network over all the images present in our dataset. We now have our pre-trained model. What is left is how we predict the location of the object during test time and what online updates are made to our model.

## CONCLUSION

**The CNNs are trained beforehand** and used to extract features, while the **last layers can quickly be trained online**. This creates a multi-domain CNN that can be used in many different scenarios, capable of discriminating between background and target. But, **the background of one video could be the target of a different video**, and so the CNN must have some method of discriminating between these two situations.

**MDNet** handles possible confusion from similar targets and backgrounds by dividing the network into **two portions**, a **shared portion** and a **portion that remains independent for every domain**. **Every domain has its own training video**, and the network is trained for the total number of

different domains. The network is first trained over K-domains iteratively where each domain classifies between its target and background. In the  $k^{th}$  iteration, the network is updated based on a minibatch that consists of the training samples from the sequence, where only a single branch  $\text{fc6}(k \bmod k)^{th}$  is enabled.

After **training is complete**, the **layers specific to the different domains are removed** and as a result, a **feature extractor capable of interpreting any given background/object pairs** is created. During the process of inference, a binary classification layer (a single fully-connected layer,  $\text{fc6}$ ) is created by removing the domain-specific layers and adding a binary classifier.

To estimate the target state in each frame,  $N$  target candidates sampled  $\{x_1, \dots, x_N\}$  around the previous target state are evaluated using the network, and we obtain their positive scores  $f^+(x^i)$  and negative scores  $f^-(x^i)$  from the network. The optimal target state  $x^*$  is given by finding the example with the maximum positive score as:

$$x^* = \arg \max_{x^i} f^+(x^i)$$

**MDNet is one of the most accurate deep learning based online training, detection free, single object tracker.**

## Deep Simple Online and Realtime Tracking (DeepSORT)

To understand **DeepSORT**, first let's see how the SORT algorithm works.

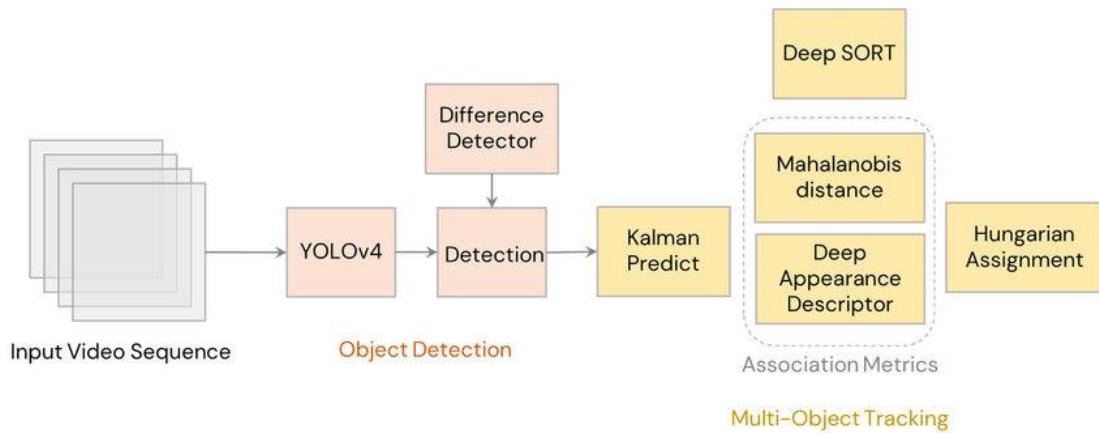
### SIMPLE ONLINE REALTIME TRACKING (SORT)

SORT is made of 4 key components which are as follows:

- **Detection:** This is the **first step** in the **tracking** module. In this step, an object detector **detects the objects in the frame** that are to be tracked. These **detections** are then **passed** on to the next step. Detectors like FrRCNN, YOLO, and more are most frequently used.
- **Estimation:** In this step, we **propagate the detections** from the current frame to the next which is **estimating the position of the target** in the next frame **using a constant velocity model**. When a detection is associated with a target, the detected bounding box is used to update the target state where the velocity components are optimally solved via the **Kalman filter** framework.
- **Data association:** We **now** have the **target bounding box** and the **detected bounding box**. So, a **cost matrix is computed** as the **intersection-over-union (IOU) distance between each detection and all predicted bounding boxes from the existing targets**. The assignment is solved optimally using the Hungarian algorithm. If the IOU of detection and target is **less than a certain threshold** value called IOUmin then that **assignment is rejected**. This technique **solves the occlusion problem** and helps maintain the IDs.
- **Creation and Deletion of Track Identities:** This module is **responsible for the creation and deletion of IDs**. Unique identities are created and destroyed according to the IOUmin. If the overlap of detection and target is less than IOUmin then it signifies the untracked object. **Tracks are terminated if they are not detected for TLost frames**, you can specify what the amount of frame should be for TLost. Should an object reappear, tracking will implicitly resume under a new identity.

The objects can be successfully tracked using SORT algorithms beating many State-of-the-art algorithms. The detector gives us detections, Kalman filters give us tracks and the Hungarian algorithm performs data association. So, Why do we even need DeepSORT? Let's look at it in the next section.

## DEEPSORT



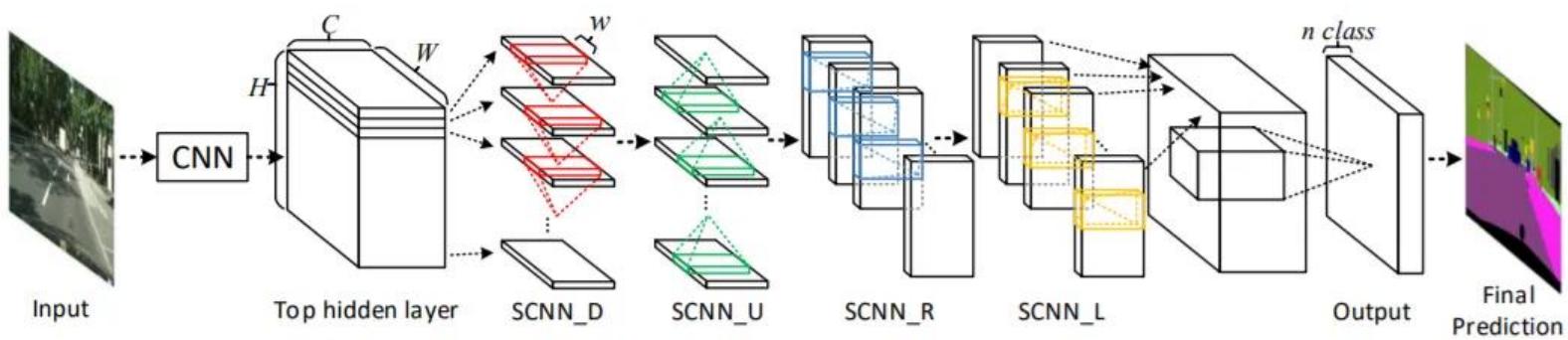
SORT performs very well in terms of tracking precision and accuracy. But SORT returns tracks with a **high number of ID switches and fails in case of occlusion**. This is because of the **association matrix** used. DeepSORT uses a **better association metric** that combines both **motion** and **appearance descriptors**. DeepSORT can be defined as the **tracking algorithm which tracks objects not only based on the velocity and motion of the object but also the appearance of the object**.

For the above purposes, a **well-discriminating feature embedding is trained offline just before implementing tracking**. The network is trained on a large-scale person re-identification dataset **making it suitable for tracking context**. To train the deep association metric model in the DeepSORT **cosine metric** learning approach is used. According to DeepSORT's paper, "The cosine distance considers appearance information that is particularly useful to recover identities after long-term occlusions when motion is less discriminative." That means cosine distance is a metric that **helps the model recover identities in case of long-term occlusion** and motion estimation also fails. Using these simple things can make the tracker even more powerful and accurate.

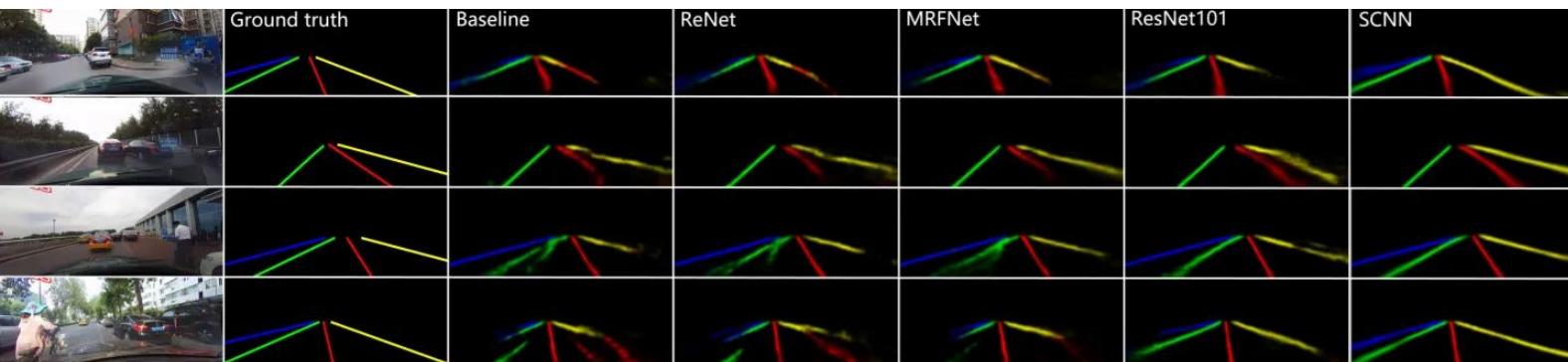
The following tool is for **Line Detection**.

## Spatial CNN (SCNN)

Spatial CNN (SCNN) proposes an architecture which “**generalizes traditional deep layer-by-layer convolutions to slice-by slice convolutions within feature maps**”. What does this mean? In a traditional layer-by-layer CNN, each convolution layer receives input from its preceding layer, applies convolutions and nonlinear activation, and sends the output to the succeeding layer. SCNN takes this a step further by **treating individual feature map rows and columns as the “layers”**, applying the same process sequentially (where sequentially means that a slice passes information to the succeeding slice only after it has received information from the preceding slices), allowing message passing of pixel information between neurons within the same layer, effectively **increasing emphasis on spatial information**.

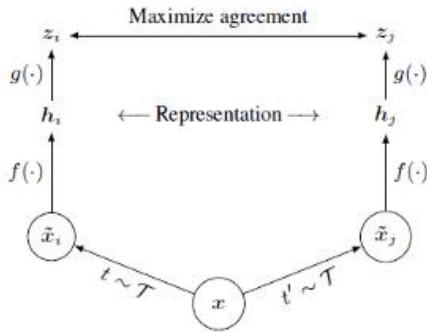


Results:



The following are tools for **contrastive learning**.

## Simple Framework for Contrastive Learning of Visual Representations (SimCLR)



**Algorithm 1** SimCLR's main learning algorithm.

```

input: batch size  $N$ , constant  $\tau$ , structure of  $f, g, \mathcal{T}$ .
for sampled minibatch  $\{x_k\}_{k=1}^N$  do
    for all  $k \in \{1, \dots, N\}$  do
        draw two augmentation functions  $t \sim \mathcal{T}, t' \sim \mathcal{T}$ 
        # the first augmentation
         $\tilde{x}_{2k-1} = t(x_k)$ 
         $h_{2k-1} = f(\tilde{x}_{2k-1})$  # representation
         $z_{2k-1} = g(h_{2k-1})$  # projection
        # the second augmentation
         $\tilde{x}_{2k} = t'(x_k)$ 
         $h_{2k} = f(\tilde{x}_{2k})$  # representation
         $z_{2k} = g(h_{2k})$  # projection
    end for
    for all  $i \in \{1, \dots, 2N\}$  and  $j \in \{1, \dots, 2N\}$  do
         $s_{i,j} = z_i^\top z_j / (\|z_i\| \|z_j\|)$  # pairwise similarity
    end for
    define  $\ell(i, j)$  as  $\ell(i, j) = -\log \frac{\exp(s_{i,j}/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(s_{i,k}/\tau)}$ 
     $\mathcal{L} = \frac{1}{2N} \sum_{k=1}^N [\ell(2k-1, 2k) + \ell(2k, 2k-1)]$ 
    update networks  $f$  and  $g$  to minimize  $\mathcal{L}$ 
end for
return encoder network  $f(\cdot)$ , and throw away  $g(\cdot)$ 

```

SimCLR learns representations by **maximizing agreement between differently augmented views of the same data example via a contrastive loss in the latent space**, as shown above.

SimCLR takes the input reference image  $x$  and produces two different views  $\tilde{x}_i$  and  $\tilde{x}_j$  through randomly sampled augmentation operations  $t$  and  $t'$ , respectively. It has been shown that having a large set of possible augmentations is crucial for this method to perform well. Then it runs the network  $f(\cdot)$  (that we are interested in) to produce representations  $h_i$  and  $h_j$ . Additionally, SimCLR uses a projection network  $g(\cdot)$  (which we are not interested in) to project features to a space (i.e.  $z_i$  and  $z_j$ ) where contrastive learning is applied. Note that the representation of  $z_i$  and  $z_j$  is used to maximize agreement based on the cosine similarity in the following way:

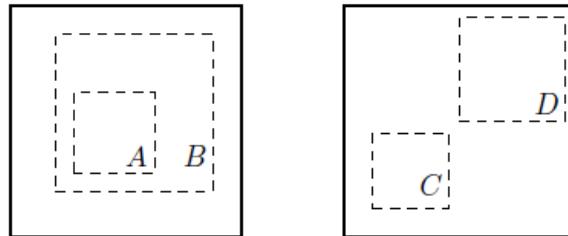
$$s(z_i, z_j) = \frac{z_i^\top z_j}{\|z_i\| \|z_j\|}$$

However, the representation of  $h_i$  and  $h_j$  is what we are actually interested in. As such,  $g(\cdot)$  and the representations of  $z_i$  and  $z_j$  only serve for better training of the model. Network  $g(\cdot)$  will be discarded after training. Another advantage of using the projection network  $g(\cdot)$  is that we can project into arbitrarily large dimensions, which provides useful flexibility.

### DATA AUGMENTATION

A stochastic data augmentation module that transforms any given data example randomly resulting in **two correlated views of the same example**, denoted  $\tilde{x}_i$  and  $\tilde{x}_j$ , as **positive pairs**.

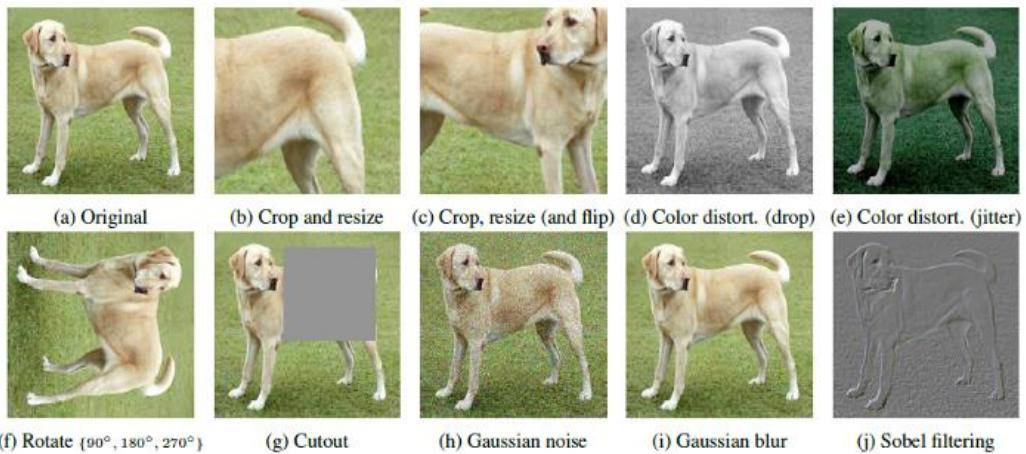
**Three simple augmentations** are applied sequentially: **random cropping followed by resize back to the original size, random color distortions, and random Gaussian blur.**



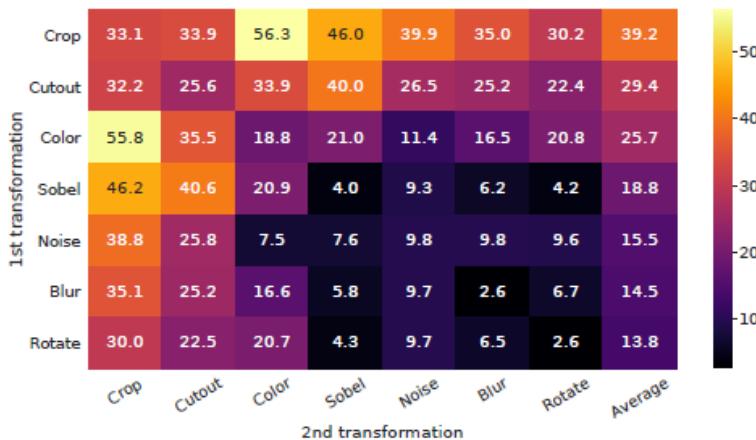
(a) Global and local views.

(b) Adjacent views.

By randomly cropping images, the contrastive prediction tasks are sampled that include **global to local view ( $B \rightarrow A$ )** or **adjacent view ( $D \rightarrow C$ ) prediction.**



The above data augmentation operators are studied.



No single transformation suffices to learn good representations.

**The combination of random crop and color distortion is crucial** to achieve a good performance.

### BASE ENCODER

A **neural network base encoder  $f()$**  that **extracts representation vectors** from augmented data examples.

ResNet is used to obtain  $h_i = f(\tilde{x}_i) = \text{ResNet}(\tilde{x}_i)$  where  $h_i$  is  $d$ -dimensional, which is the output after the average pooling layer.

**Unsupervised learning benefits more from bigger models than its supervised counterpart.**

## PROJECTION HEAD

A small neural network **projection head  $g()$**  that **maps representations to the space where contrastive loss is applied**.

A **MLP with one hidden layer** is used to obtain  $z_i$ :

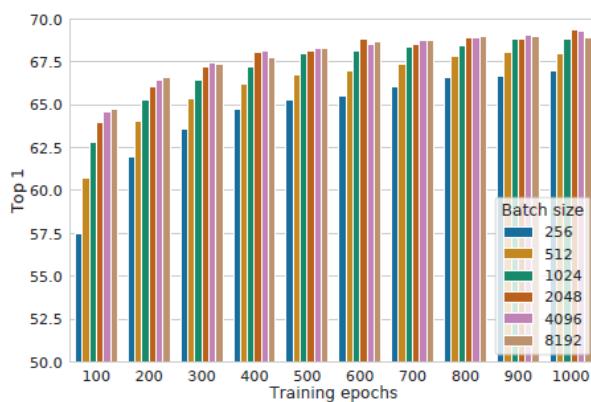
$$z_i = g(h_i) = W^{(2)}\sigma(W^{(1)}h_i)$$

Where  $\sigma$  is a ReLU nonlinearity.

It is found **that it is beneficial to define the contrastive loss on  $z_i$ 's rather than  $h_i$ 's**.

## CONTRASTIVE LOSS

A **minibatch of  $N$  examples** is randomly sampled.



**Large batch size** is beneficial, **longer training time** is also beneficial.

The **contrastive prediction task** is defined on pairs of augmented examples derived from the minibatch, resulting in  **$2N$  data points**.

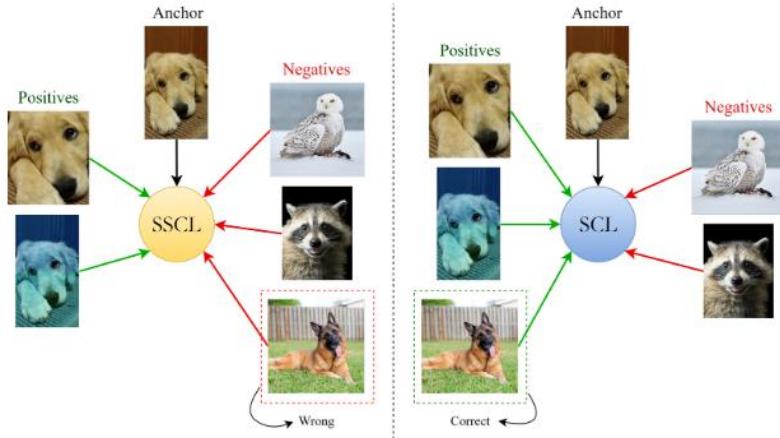
**Given a positive pair, the other  $2(N-1)$  augmented examples within a minibatch as negative examples.**

**The loss function for a positive pair** of examples  $(i, j)$  is defined as:

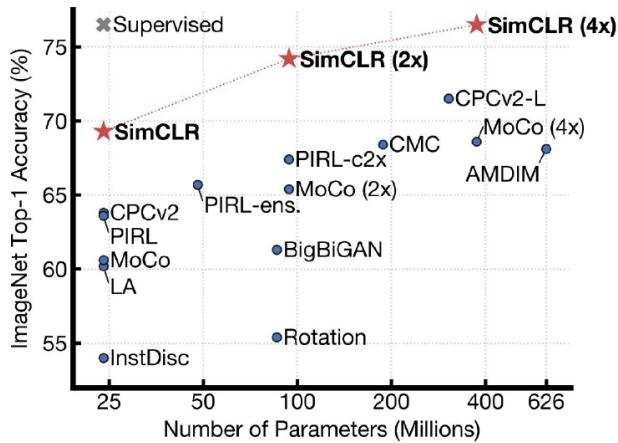
$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(z_i, z_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(z_i, z_k)/\tau)},$$

where **sim()** is **cosine similarity**,  $\tau$  is the **temperature** parameter.

The final loss is computed across all positive pairs, both  $(i, j)$  and  $(j, i)$ , in a mini-batch.



## RESULTS



Train feature encoder on **ImageNet** (entire training set) using SimCLR.

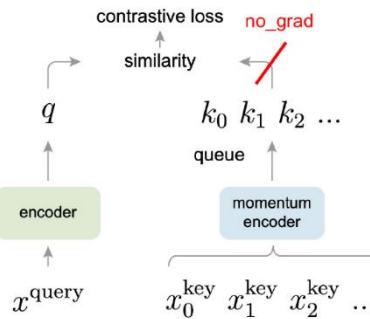
Freeze feature encoder, train a linear classifier on top with labeled data.

When looking at Top-1 accuracy on ImageNet, the SimCLR method performs surprisingly well compared to other self-supervised methods. When increasing the number of parameters it is even possible to achieve results that are on par with supervised pre-trained models. When the feature encoder is pre-trained on the entire ImageNet data set without labels using SimCLR and subsequently fine-tuned only on 1% or 10% of the ImageNet data set with labels, the method even performs significantly better than a supervised model, that was trained on only 1% or 10% of the data set with labels.

A drawback of SimCLR is that it **requires large batch sizes** (between 2048 and 4096) to perform well. Due to the kind of architecture (i.e. Siamese network) this leads to a huge memory requirement. Therefore, it requires distributed training on TPUs (i.e. training on GPUs not possible).

## Momentum Contrast (MoCo)

Momentum Contrast is a method that is similar to SimCLR, but includes major innovations that aim to alleviate the large memory requirements of SimCLR. The figure below shows an overview of this method. The crucial thing about Momentum Contrast is to keep a running queue - or dictionary - of keys for negative samples (i.e. a ring buffer of mini-batches). Gradients are then only back-propagated to the query encoder and not to the queue of keys.



Therefore, we do not have to store intermediate representations, but we only have to store the final result (i.e. the keys). For this reason, the dictionary can be much larger than the mini-batch size. However, if the same encoder that is used for the query would also be used for the keys, then this would lead to inconsistencies due to the fact that the query encoder changes during training (i.e. through backpropagation). To improve the consistency of the keys, the authors proposed a momentum update rule:

$$\theta_k = \beta\theta_k + (1 - \beta)\theta_q$$

The momentum encoder thus is a linear combination of the query encoder  $\theta_q$  and the previous momentum encoder  $\theta_k$ . This is a key improvement to make the method work also with smaller mini-batch sizes.

An improved version of this method - called MoCo v2 - uses stronger data augmentation techniques as well as a non-linear projection head. These two additions turned out to be crucial as well. Using these two additions MoCo v2 outperforms SimCLR while using much smaller mini-batch sizes (i.e. 256). Furthermore, MoCo v2 can be trained on a regular 8 × V 100 GPU node, instead of TPUs.

The dictionary is dynamic in the sense that the keys are randomly sampled, and that the key encoder evolves during training.

The hypothesis is that **good features can be learned by a large dictionary that covers a rich set of negative samples, while the encoder for the dictionary keys is kept as consistent as possible despite its evolution.**

## DICTIONARY AS A QUEUE

MoCo maintains the dictionary as a queue of data samples. This allows to reuse the encoded keys from the immediate preceding mini-batches. The dictionary size can be much larger than a typical mini-batch size.

The samples in the dictionary are progressively replaced. **The current mini-batch is enqueued to the dictionary, and the oldest mini-batch in the queue is removed.**

**Removing the oldest mini-batch** can be **beneficial**, because its encoded keys are the **most outdated**.

## MOMENTUM UPDATE

Using a queue makes it intractable to update the key encoder by back-propagation. A **naïve solution** is to **copy the key encoder  $f_k$  from the query encoder  $f_q$** , ignoring this gradient. But this solution yields **poor results**. It is hypothesized that such failure is caused by the **rapidly changing encoder that reduces the key representations' consistency**.

Formally, denoting the parameters of  $f_k$  as  $k$  and those of  $f_q$  as  $q$ ,  $k$  is updated by:

$$\theta_k \leftarrow m\theta_k + (1 - m)\theta_q.$$

Here  $m \in [0, 1]$  is a **momentum coefficient**. Only the parameters  $q$  are updated by back-propagation.

As a result, though the keys in the queue are encoded by different encoders (in different mini-batches), **the difference among these encoders can be made small**. In experiments, a **relatively large momentum** (e.g.,  $m=0.999$ , default) works much better than a smaller value (e.g.,  $m=0.9$ ), suggesting that a slowly evolving key encoder is a core to making use of a queue.

## RESULTS

case	MLP	aug+	unsup. cos	pre-train epochs	batch	ImageNet acc.
MoCo v1 [6]				200	256	60.6
SimCLR [2]	✓	✓	✓	200	256	61.9
SimCLR [2]	✓	✓	✓	200	8192	66.6
<b>MoCo v2</b>	✓	✓	✓	200	256	<b>67.5</b>

*results of longer unsupervised training follow:*

SimCLR [2]	✓	✓	✓	1000	4096	69.3
<b>MoCo v2</b>	✓	✓	✓	800	256	<b>71.1</b>

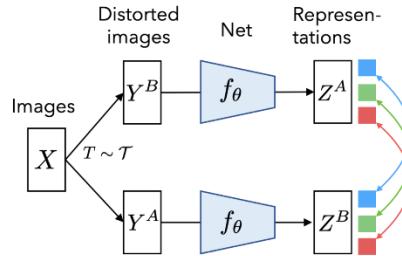
Table 2. **MoCo vs. SimCLR**: ImageNet linear classifier accuracy (**ResNet-50, 1-crop 224×224**), trained on features from unsupervised pre-training. “aug+” in SimCLR includes blur and stronger color distortion. SimCLR ablations are from Fig. 9 in [2] (we thank the authors for providing the numerical results).

### Key insights:

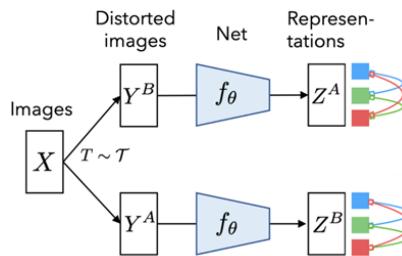
- ▶ Non-linear projection head and strong data augmentation are crucial
- ▶ Using both, MoCo v2 outperforms SimCLR with much smaller mini-batch sizes (i.e., MoCo v2 runs on a regular 8x V100 GPU node)

## Barlow Twins

Barlow Twins that was proposed quite recently deviates somewhat from the classical paradigm and makes things even simpler. It is based on information theory and tries to reduce the redundancy between neurons, instead of samples. In other words: instead of looking at distances between samples it looks at distances between neurons. The idea behind this is that neurons should be invariant to data augmentations, but independent of others.



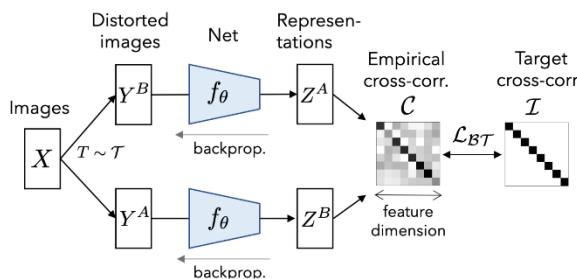
In the example above an image  $X$  is augmented into two different views  $Y^A$  and  $Y^B$  and then passed through a network  $f_\theta$  that we want to train in order to produce feature representations  $Z^A$  and  $Z^B$ . For now we will imagine that both  $Z^A$  and  $Z^B$  are three-dimensional feature vectors, one feature per channel for the RGB-channels of the image. We now want to achieve that the **features for each of the channels are similar between the two views** (the colored arrows above) because the views are two different augmentations of the same object/image (i.e.  $X$ ). However, **at the same time we want the features of each of the representations  $Z$  to be different from each other** (as in the following figure):



The idea to achieve this mathematically is to **compute the empirical cross-correlation matrix**:

$$c_{ij} = \frac{\sum_b Z_{b,i}^A Z_{b,j}^B}{\sqrt{\sum_b (Z_{b,i}^A)^2} \sqrt{\sum_b (Z_{b,j}^B)^2}}$$

and to **encourage it to become the identity matrix**:

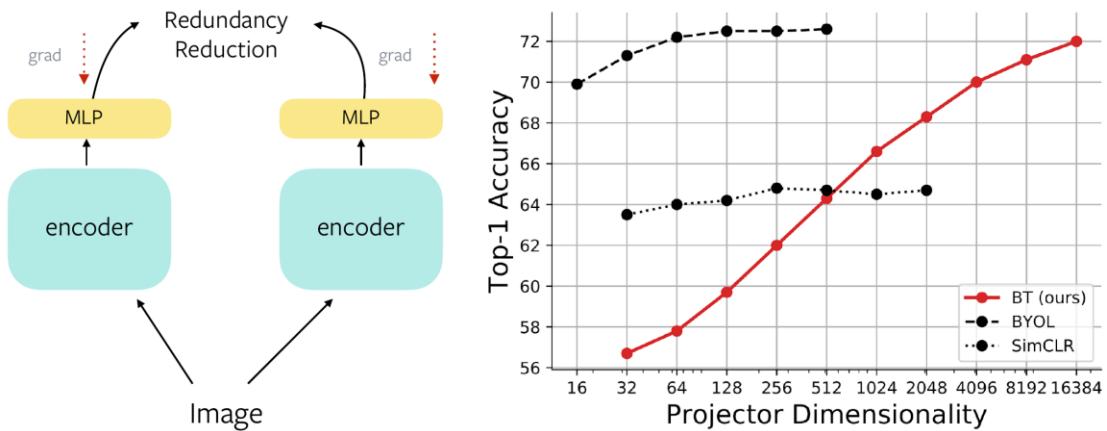


So we need a loss  $\mathcal{L}_{BT}$  which enforces the off-diagonal elements of the empirical cross correlation matrix to be zero (minimize redundancy) and the diagonal elements to be one (i.e. neurons across augmentations should be correlated):

$$\mathcal{L}_{BT} = \underbrace{\sum_i (1 - c_{ii})^2 \lambda}_{\text{Invariance term}} + \underbrace{\sum_i \sum_{j \neq i} c_{ij}^2}_{\text{redundancy reduction term}}$$

The crucial **difference to SimCLR** is that **no negative samples are needed**, because the contrastive learning happens in the neuron space, rather than the sample space.

The results of the method are on par with state-of-the-art techniques such as SimCLR, even though it is much simpler to implement. Furthermore, it is only mildly affected by the mini-batch size. Similarly to SimCLR, the authors also found that projection into higher-dimensional space during pre-training leads to better results:



By increasing the dimensionality of the projection MLP the performance of the method keeps on increasing.

The following are tools for **autoencoders**.

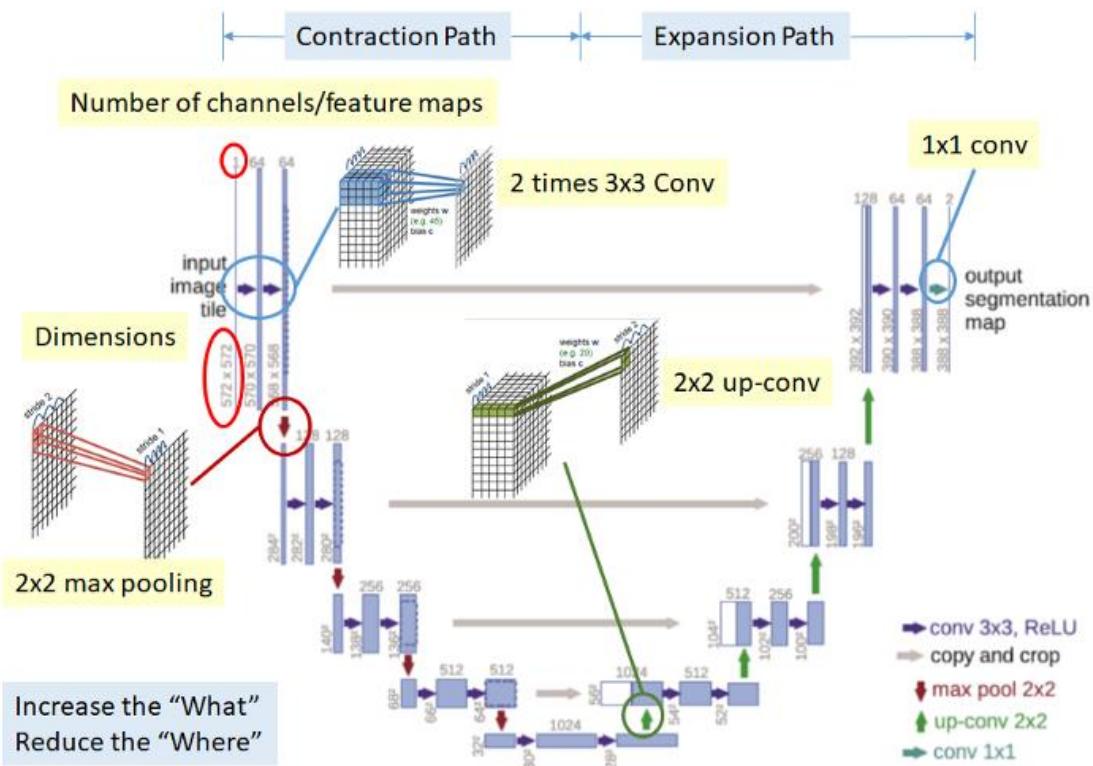
## U-NET

**U-Net** is one of the famous **Fully Convolutional Networks (FCN)** in **biomedical image segmentation**, which has been published in [2015 MICCAI](#) with more than 3000 citations.

In the field of biomedical image annotation, we always need experts, who acquired the related knowledge, to annotate each image. And they also consume large amount of time to annotate. If the annotation process becomes automatic, less human efforts and lower cost can be achieved. Or it can be act as an assisted role to reduce the human mistake.

In this paper, they segment/annotate the Electron Microscopic (EM) Image. They also modify the network a little bit to segment/annotate the dental X-ray image in [2015 ISBI](#).

### U-NET NETWORK ARCHITECTURE



The **U-net architecture** is as shown above. It consists of **contraction path** and **expansion path**.

#### Contraction Path

**Consecutive of two times of  $3 \times 3$  Convolution and  $2 \times 2$  Max Pooling** is done. This can help to **extract more advanced features** but it also **reduce the size of feature maps**.

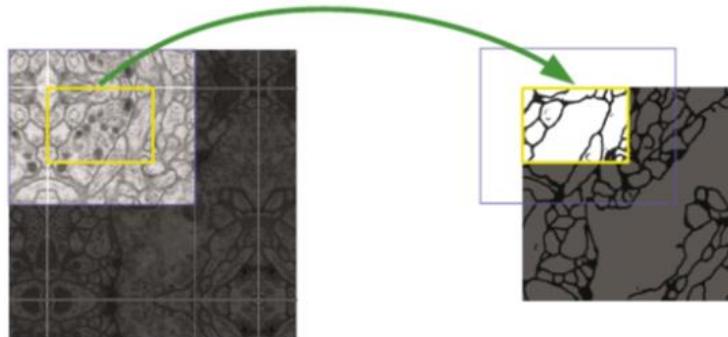
#### Expansion Path

**Consecutive of  $2 \times 2$  Up-convolution and two times of  $3 \times 3$  Convolution** is done to **recover the size of segmentation map**. However, the above process **reduces the “where”** though it **increases the “what”**. That means, we can **get advanced features**, but we also **lose the localization information**.

Thus, **after each up-convolution, we also have concatenation of feature maps** (gray arrows) that are with the same level. This helps to **give the localization information from contraction path to expansion path.**

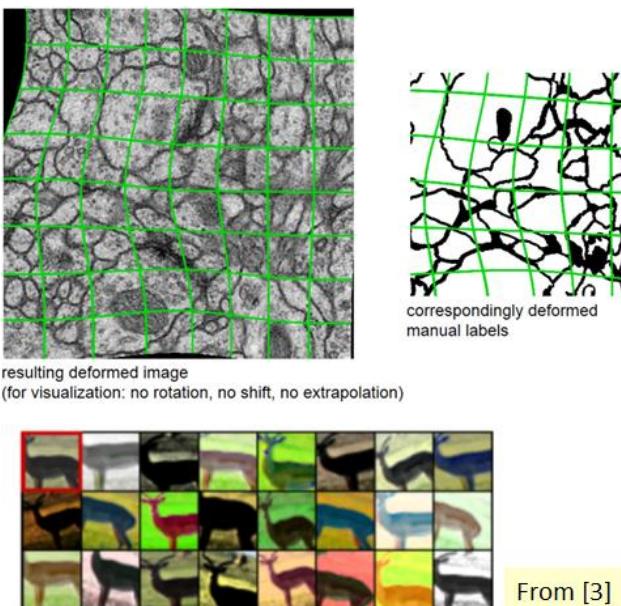
At the end,  **$1 \times 1$  convolution** to map the feature map size from 64 to 2 since the output feature map only have 2 classes, cell and membrane.

### OVERLAP TILE STRATEGY



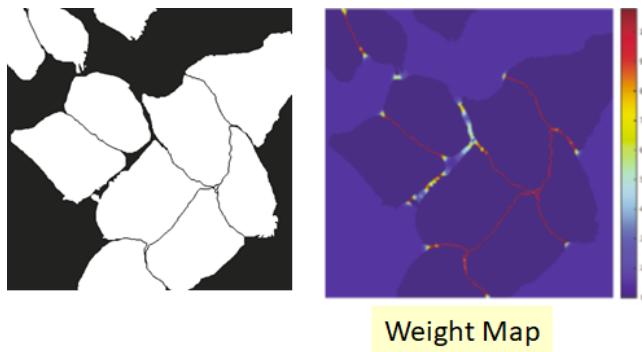
Since unpadded convolution is used, output size is smaller than input size. Instead of downsizing before network and upsampling after network, overlap tile strategy is used. Thereby, the **whole image is predicted part by part** as in the figure above. The yellow area in the image is predicted using the blue area. At the image boundary, image is extrapolated by mirroring.

### ELASTIC DEFORMATION FOR DATA AUGMENTATION



Since the training set can only be annotated by experts, the training set is small. To increase the size of training set, **data augmentation is done by randomly deforming the input image and output segmentation map.**

## SEPARATION OF TOUCHING OBJECTS



Since the touching objects are closely placed each other, they are easily merged by the network, **to separate them, a weight map is applied to the output of network.**

$$w(x) = w_c(x) + w_0 \cdot \exp\left(-\frac{(d_1(x) + d_2(x))^2}{2\sigma^2}\right)$$

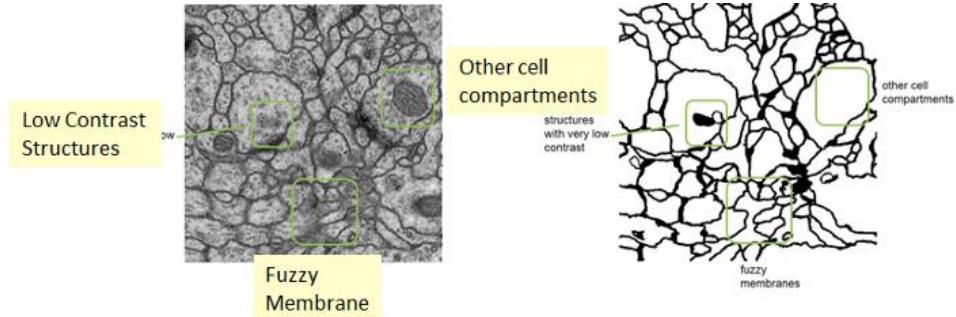
To compute the weight map as above,  $d_1(x)$  is the **distance to the nearest cell border at position  $x$** ,  $d_2(x)$  is the **distance to the second nearest cell border**. Thus, at the border, weight is much higher as in the figure.

$$p_k(x) = \frac{\exp(a_k(x))}{\sum_{k'=1}^K \exp(a_{k'}(x))}$$

$$E = \sum_{x \in \Omega} w(x) \log(p_{l(x)}(x))$$

Thus, the **cross entropy function is penalized at each position by the weight map**. And it **help to force the network to learn the small separation borders between touching cells**.

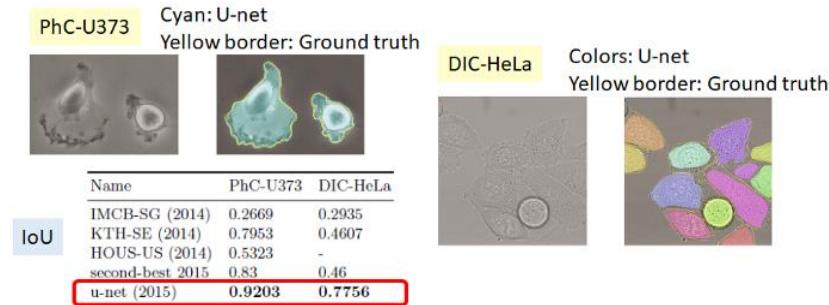
## RESULTS

*ISBI 2012 Challenge*

Rank	Group name	Warping Error	Rand Error	Pixel Error
	** human values **	0.000005	0.0021	0.0010
1.	u-net	<b>0.000353</b>	0.0382	<b>0.0611</b>
2.	DIVE-SCI	0.000355	0.0305	0.0584
3.	IDSIA [2]	0.000420	0.0504	0.0613
4.	DIVE	0.000430	0.0545	<b>0.0582</b>
:				
10.	IDSIA-SCI	0.000653	<b>0.0189</b>	0.1027

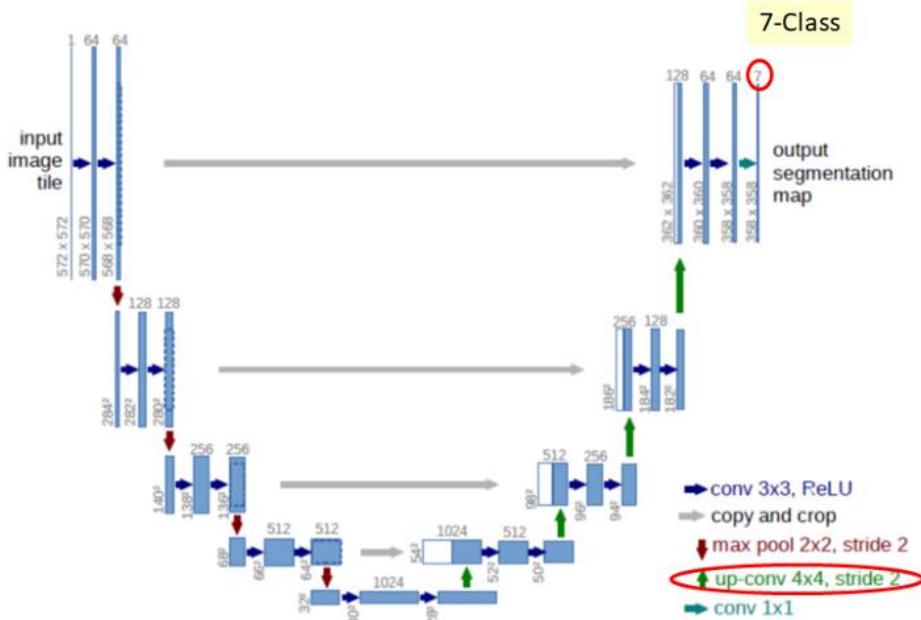
- **Warping Error:** A segmentation metric that penalizes topological disagreements.
- **Rand Error:** A measure of similarity between two clusters or segmentations.
- **Pixel Error:** A standard pixel-wise error.
- **Training time:** 10 Hours
- **Testing speed:** around 1s per image

### PhC-U373 and DIC-HeLa Datasets

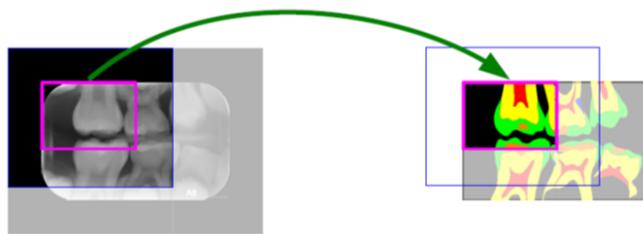


U-Net got the highest IoU for these two datasets.

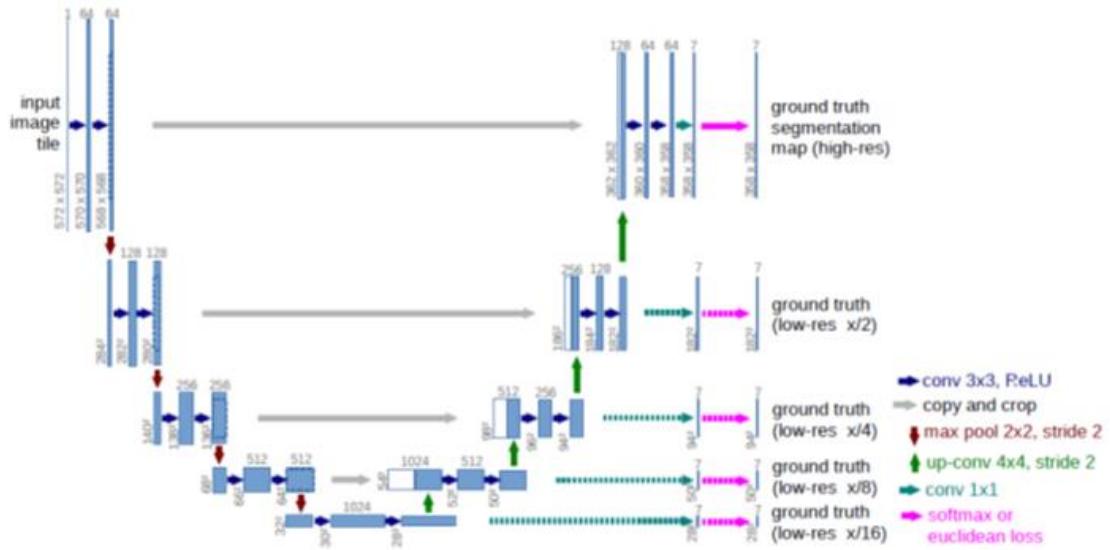
### SOME MODIFICATIONS OF U-NET (FOR DENTAL X-RAY IMAGE SEGMENTATION)



This time,  $4 \times 4$  Up-convolution is used, and  $1 \times 1$  Convolution to map feature maps from 64 to 7 because the output for each location has 7 classes.

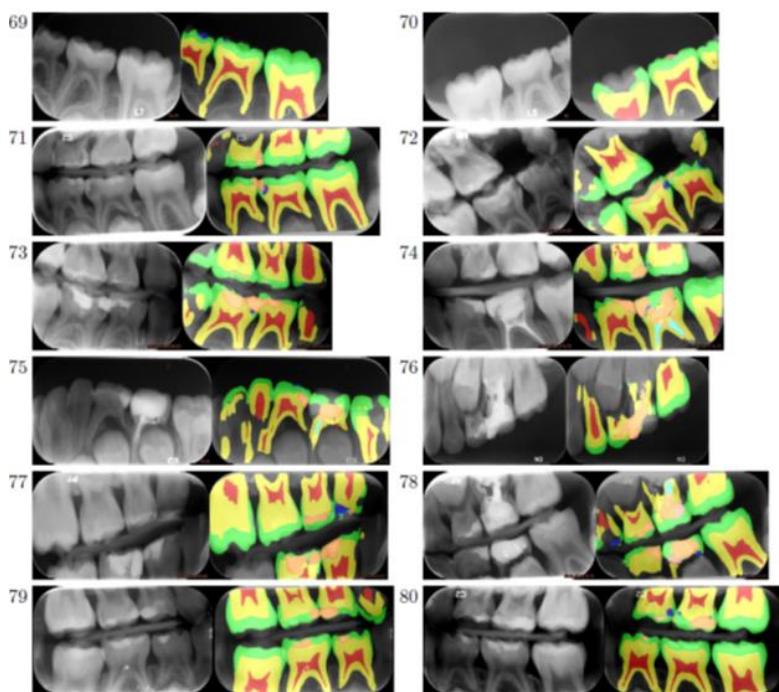


At the Overlap Tile Strategy, **zero padding** is used instead of mirroring at the image boundary.  
Because mirroring isn't making any sense for teeth.



There are **additional loss layers** to the low-resolution feature maps using softmax loss, in order to **guide the deep layers to directly learn the segmentation classes**.

## Results

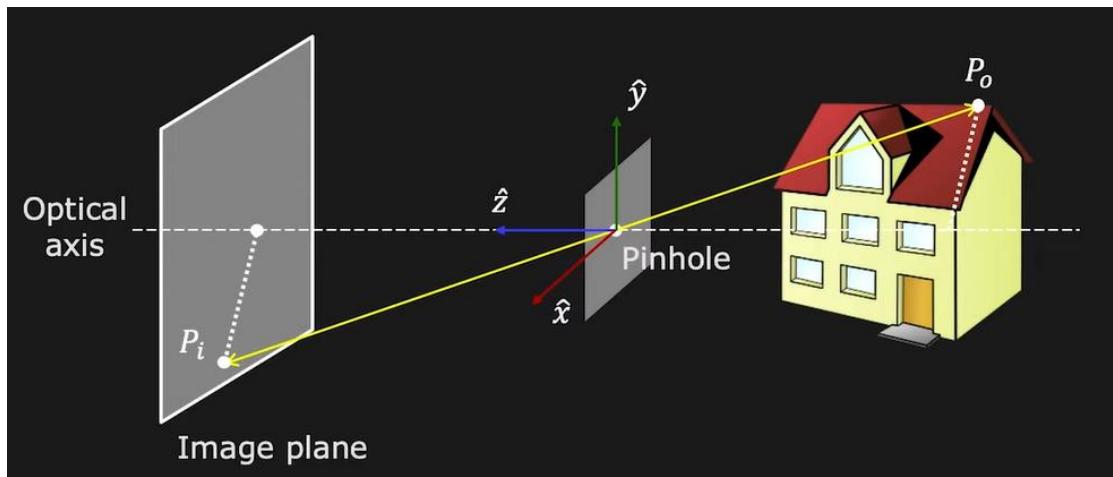


## CAMERAS

### Pinhole Camera (Camera Obscura)

Pinhole camera is the simplest tool to generate an image. And it's easy to make since its components just consist of a pitch-dark chamber, a small hole called “**Pinhole**” and a scene. To form the image, the light passing through the pinhole and incident to the scene. This makes the image smaller and inverted.

The geometric properties of pinhole camera are quite easy as shown below in the figure.



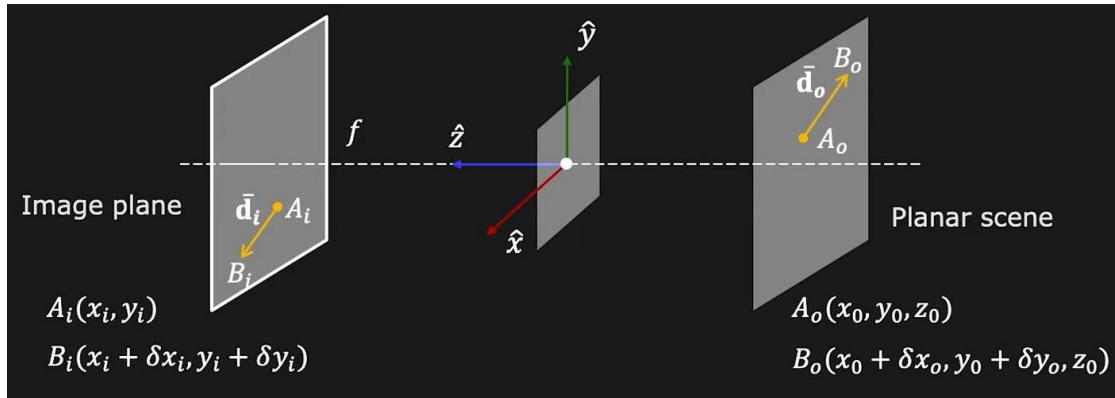
From the figure, the object located at point  $P_o$ , at a horizontal distance  $z_0$  from the pinhole and vertical distance  $y_0$  from the **optical axis**. And  $f$  is the distance between the pinhole and the scene, called **effective focal length**. Such that:

$$\begin{aligned} r_o &= (x_0, y_0, z_0) \\ r_i &= (x_i, y_i, f) \end{aligned}$$

We found the relationship between object coordinates  $r_o$  and image coordinates  $r_i$  and focal length  $f$  by:

$$\frac{r_i}{f} = \frac{r_o}{z_0} \Rightarrow \begin{cases} \frac{x_i}{f} = \frac{x_0}{z_0} \\ \frac{y_i}{f} = \frac{y_0}{z_0} \end{cases}$$

## IMAGE MAGNIFICATION



The **magnification** is defined as:

$$|m| = \frac{\|\bar{d}_i\|}{\|\bar{d}_0\|} = \frac{\sqrt{\delta x_i^2 + \delta y_i^2}}{\sqrt{\delta x_0^2 + \delta y_0^2}} \Rightarrow$$

From perspective projection we got:

$$\begin{cases} \frac{x_i}{f} = \frac{x_0}{z_0} \\ \frac{y_i}{f} = \frac{y_0}{z_0} \\ \frac{x_i + \delta x_i}{f} = \frac{x_0 + \delta x_0}{z_0} \\ \frac{y_i + \delta y_i}{f} = \frac{y_0 + \delta y_0}{z_0} \end{cases} \Rightarrow \begin{cases} \frac{\delta x_i}{f} = \frac{\delta x_0}{z_0} \\ \frac{\delta y_i}{f} = \frac{\delta y_0}{z_0} \end{cases}$$

Plugging it into the magnification:

$$\Rightarrow |m| = \frac{\|\bar{d}_i\|}{\|\bar{d}_0\|} = \frac{\sqrt{\delta x_i^2 + \delta y_i^2}}{\sqrt{\delta x_0^2 + \delta y_0^2}} = \frac{\sqrt{\delta \left(\frac{f}{z_0} x_0\right)^2 + \delta \left(\frac{f}{z_0} y_0\right)^2}}{\sqrt{\delta x_0^2 + \delta y_0^2}} = \frac{\sqrt{\left(\frac{f}{z_0}\right)^2 (\delta x_0^2 + \delta y_0^2)}}{\sqrt{\delta x_0^2 + \delta y_0^2}} = \left| \frac{f}{z_0} \right|$$

Where  $m$  is negative when image is **inverted**, and the image size is **inversely proportional** to depth.

**Remarks:**

- $m$  can be assumed to be **constant** if the range of scene depth  $\Delta z$  is much smaller than the average scene depth  $\tilde{z}$
- Ratio of areas:

$$\frac{\text{Area}_i}{\text{Area}_0} = m^2$$

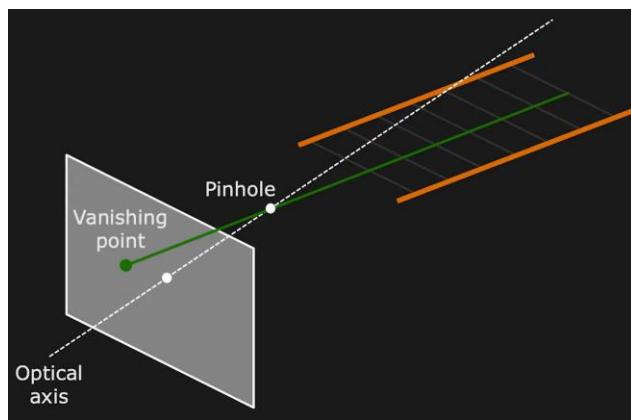
## VANISHING POINT



Location of **Vanishing Point** depends on the **orientation** of parallel straight lines.

### *Finding the Vanishing Point*

Assuming we have a set of parallel lines and we want to find the vanishing point corresponding to those lines. We need to construct a line that is parallel to these two lines and passes through the pinhole, wherever that line pierces the image this is the vanishing point.



Calculating the coordinates

Vanishing point of the line is the projection of point  $P$ :

$$(x_{vp}, y_{vp}) = \left( f \frac{l_x}{l_z}, f \frac{l_y}{l_z} \right)$$

Where the direction of the parallel lines (in 3D) is given by:

$$\hat{l} = (l_x, l_y, l_z)$$

And we create a point  $P$  which is in the direction above from the pinhole of the camera:

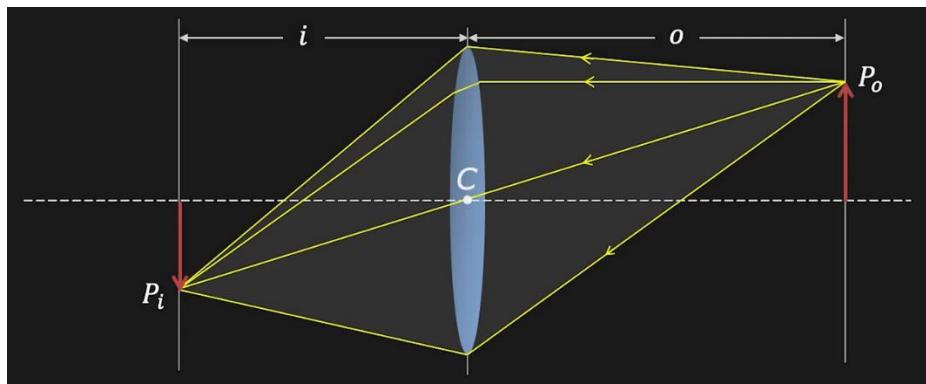
$$P = (l_x, l_y, l_z)$$

However, the pinhole camera is not practical to use as expected. The smaller hole produces the sharper image (but if it's too small it will cause **diffraction**) and it comes along with longer exposure time (to capture bright images). We can calculate the ideal pinhole diameter by:

$$d \approx 2\sqrt{f\lambda}$$

Where  $f$  is the effective focal length and  $\lambda$  is the wavelength.

So the **optical lens model** is used for these reasons. For use, we need to replace the pinhole with a thin lens. We assumed that it is infinitely thin. So the geometry of the image is similar to the image from the pinhole camera. The geometric properties of the thin lens model are shown in this figure.



By lens law:

$$\frac{1}{i} + \frac{1}{o} = \frac{1}{f} \xrightarrow{o=\infty} f = i$$

The magnification:

$$m = \frac{h_i}{h_0} = \frac{i}{o}$$

## Mathematical Notations

### HOMOGENEOUS COORDINATES

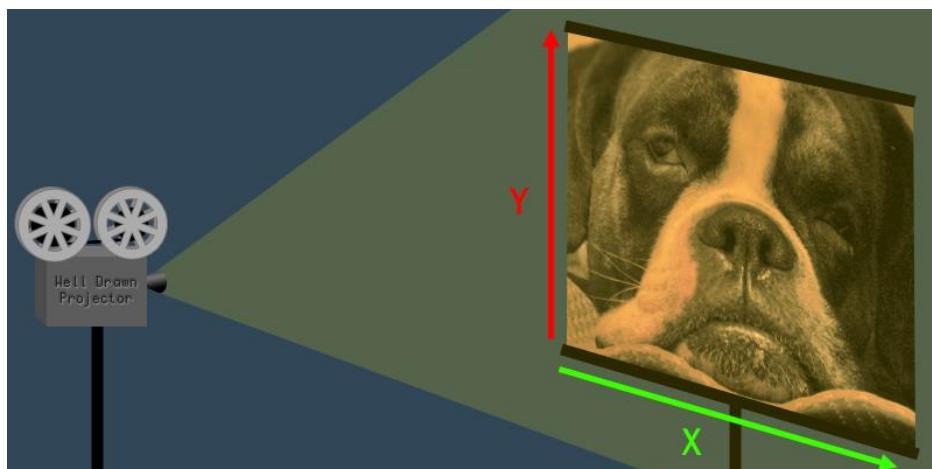
Most of the time when working with 3D, we are thinking in terms of Euclidean geometry – that is, coordinates in three-dimensional space ( $X$ ,  $Y$ , and  $Z$ ). However, there are certain situations where it is useful to think in terms of projective geometry instead. **Projective geometry** has an extra dimension, called  $W$ , in addition to the  $X$ ,  $Y$ , and  $Z$  dimensions. This four-dimensional space is called “**projective space**” ( $\mathbb{P}^2 = \mathbb{R}^2 \setminus \{(0,0,0)\}$ ), and coordinates in projective space are called “**homogeneous coordinates**”.

For the purposes of 3D software, the terms “**projective**” and “**homogeneous**” are basically interchangeable with “4D”.

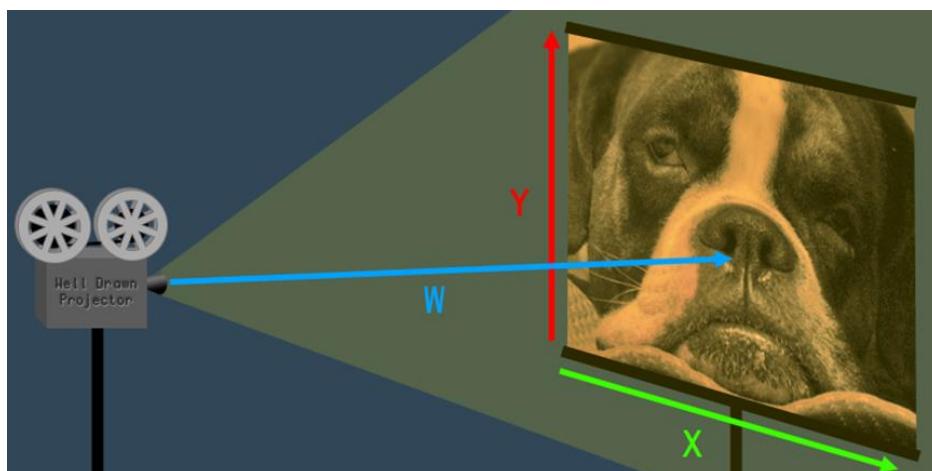
### An Analogy in 2D

First, let's look at how projective geometry works in 2D, before we move on to 3D.

Imagine a projector that is projecting a 2D image onto a screen. It's easy to identify the  $X$  and  $Y$  dimensions of the projected image:



Now, if you step back from the 2D image and look at the projector and the screen, you can see the  $W$  dimension too. The  $W$  dimension is the distance from the projector to the screen.



So what does the  $W$  dimension do, exactly? Imagine what would happen to the 2D image if you increased or decreased  $W$  – that is, if you increased or decreased the distance between the projector and the screen. If you move the projector closer to the screen, the whole 2D image becomes smaller. If you move the projector away from the screen, the 2D image becomes larger. As you can see, the value of  $W$  affects the size (a.k.a. **scale**) of the image. Homogeneous vectors are considered as equivalent when they differ only up to scale.



An **inhomogeneous vector**  $x$  is converted to a **homogeneous vector**  $\tilde{x}$  as follows:

$$\tilde{x} = \begin{pmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{w} \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \tilde{x}$$

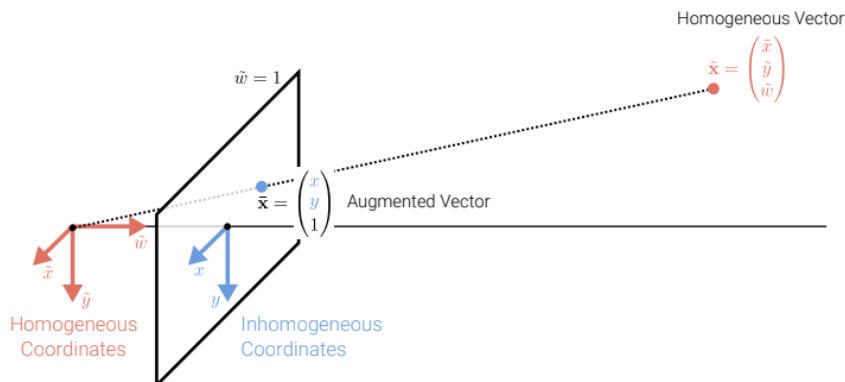
with **augmented vector**  $\tilde{x}$ . We say augmented vector  $\tilde{x}$  for all homogeneous vectors which last coordinate is equal to 1.

To convert in the opposite direction, we divide by the last element  $\tilde{w}$ :

$$\tilde{x} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ \tilde{w} \end{pmatrix} = \frac{1}{\tilde{w}} \tilde{x} = \frac{1}{\tilde{w}} \begin{pmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{w} \end{pmatrix} = \begin{pmatrix} \frac{\tilde{x}}{\tilde{w}} \\ \frac{\tilde{y}}{\tilde{w}} \\ 1 \end{pmatrix}$$

Homogeneous points whose last element is  $\tilde{w} = 0$  are called **ideal points** or **points at infinity**.

These points can't be represented with inhomogeneous coordinates as we can't divide by  $\tilde{w}$ .



## 2D Lines

**2D lines** can also be expressed using homogeneous coordinates  $\tilde{l} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$ :

$$\{\tilde{x} | \tilde{l}^T \tilde{x} = 0\} \Leftrightarrow \{x, y | ax + by + c = 0\}$$

We get the line equation by the inner product of the vector  $\tilde{l}$  and an augmented vector. For all  $(x, y)$  for which the equation is zero, the points are located on the line. We can **normalize**  $\tilde{l}$  so

that  $\tilde{l} = \begin{pmatrix} n_x \\ n_y \\ d \end{pmatrix} = \begin{pmatrix} n \\ d \end{pmatrix}$  with  $\|n\|_2 = 1$ . In this case,  $n$  is the **normal vector** perpendicular to the

line and  $d$  is its **distance to the origin**.

An exception is the line at infinity  $\tilde{l}_\infty = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$  which passes through all ideal points.

The **intersection** of two lines is given by:

$$\tilde{x} = \tilde{l}_1 \times \tilde{l}_2$$

*Constructive Proof*

Let's define:

$$\tilde{l}_1 = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}, \quad \tilde{l}_2 = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

We know that the sets of points lying on  $\tilde{l}_1$  or  $\tilde{l}_2$  are given by:

$$\begin{aligned} & \{x, y | a_1x + a_2y + a_3 = 0\} \\ & \{x, y | b_1x + b_2y + b_3 = 0\} \end{aligned}$$

As a result, we obtain the system of two linear equations:

$$\begin{cases} a_1x + a_2y + a_3 = 0 \\ b_1x + b_2y + b_3 = 0 \end{cases}$$

When solving the first equation for  $x$ , we get:

$$x = -\frac{a_2}{a_1}y - \frac{a_3}{a_1}$$

Let's insert this in our second equation:

$$\begin{aligned} & -\frac{b_1a_2}{a_1}y - \frac{a_3b_1}{a_1} + b_2y + b_3 = 0 \Rightarrow y \left( b_2 - \frac{b_1a_2}{a_1} \right) = \frac{a_3b_1}{a_1} - b_3 \Rightarrow \\ & \Rightarrow y \left( \frac{b_2a_1 - b_1a_2}{a_1} \right) = \frac{a_3b_1}{a_1} - b_3 \Rightarrow y = \frac{a_3b_1 - b_3a_1}{b_2a_1 - b_1a_2} \end{aligned}$$

In a similar fashion, we solve the first equation for  $y$ :

$$y = -\frac{a_1}{a_2}x - \frac{a_3}{a_2}$$

and insert this in the second equation:

$$\begin{aligned} b_1x - \frac{b_2a_1}{a_2}x - \frac{b_2a_3}{a_2} + b_3 = 0 &\Rightarrow x\left(b_1 - \frac{b_2a_1}{a_2}\right) = \frac{b_2a_3}{a_2} - b_3 \Rightarrow \\ &\Rightarrow x\left(\frac{b_1a_2 - b_2a_1}{a_2}\right) = \frac{b_2a_3 - b_3a_2}{a_2} \Rightarrow x = \frac{b_3a_2 - b_2a_3}{b_2a_1 - b_1a_2} \end{aligned}$$

We find that the intersection point in heterogenous coordinates is given by:

$$x_{\text{intersect}} = \begin{pmatrix} b_3a_2 - b_2a_3 \\ b_2a_1 - b_1a_2 \\ a_3b_1 - b_3a_1 \\ b_2a_1 - b_1a_2 \end{pmatrix}$$

When we now solve the cross product of the two lines in homogeneous coordinates:

$$\tilde{x}_{\text{intersect}} = \tilde{l}_1 \times \tilde{l}_2 = \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix} \sim \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_1b_2 - a_2b_1 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \\ 1 \end{pmatrix}$$

Hence we can see:

$$\tilde{x}_{\text{intersect}} = \begin{pmatrix} x_{\text{intersect}} \\ 1 \end{pmatrix}$$

Similarly, the **line joining two points** can be compactly written as:

$$\tilde{l} = \tilde{x}_1 \times \tilde{x}_2$$

*Constructive Proof*

The inner product can also be used to verify that  $\tilde{x}_1$  and  $\tilde{x}_2$  both lie on the line  $\tilde{l} = \tilde{x}_1 \times \tilde{x}_2$ . For this, let:

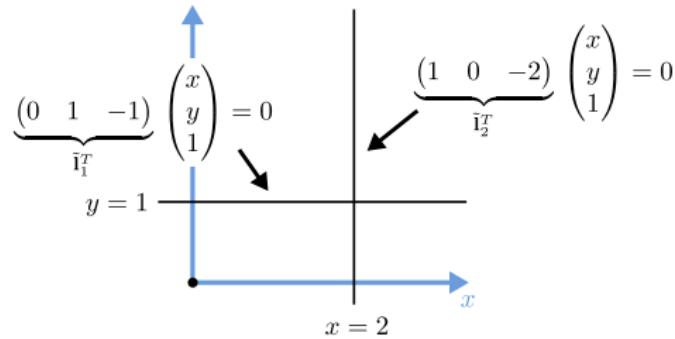
$$\tilde{x}_1 = \begin{pmatrix} \tilde{x}_1^1 \\ \tilde{x}_1^2 \\ \tilde{x}_1^3 \end{pmatrix}, \quad \tilde{x}_2 = \begin{pmatrix} \tilde{x}_2^1 \\ \tilde{x}_2^2 \\ \tilde{x}_2^3 \end{pmatrix}$$

For example, for  $\tilde{x}_1$  we see:

$$\tilde{l}^T \cdot \tilde{x}_1 = \begin{pmatrix} \tilde{x}_1^2\tilde{x}_2^3 - \tilde{x}_1^3\tilde{x}_2^2 \\ \tilde{x}_1^3\tilde{x}_2^1 - \tilde{x}_1^1\tilde{x}_2^3 \\ \tilde{x}_1^1\tilde{x}_2^2 - \tilde{x}_1^2\tilde{x}_2^1 \end{pmatrix}^T \begin{pmatrix} \tilde{x}_1^1 \\ \tilde{x}_1^2 \\ \tilde{x}_1^3 \end{pmatrix} = 0$$

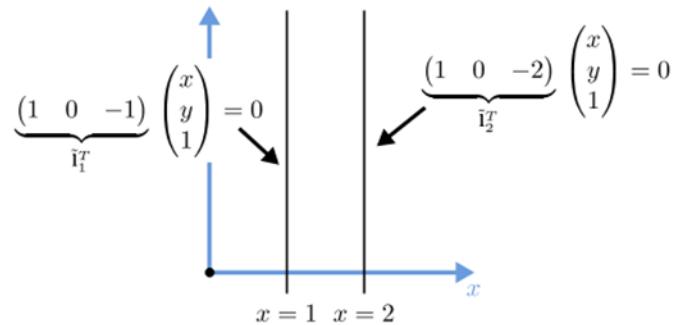
And similarly, we can show that  $\tilde{l}^T \cdot \tilde{x}_2 = 0$ . As both points  $\tilde{x}_1$  and  $\tilde{x}_2$  lie on the line  $\tilde{l}$ ,  $\tilde{l}$  is the line going through the two points.

*Example*



$$\tilde{l}_1 \times \tilde{l}_2 = [\tilde{l}_1]_x \tilde{l}_2 = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix} = \begin{pmatrix} -2 \\ -1 \\ -1 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}$$

Or:



$$\tilde{l}_1 \times \tilde{l}_2 = [\tilde{l}_1]_x \tilde{l}_2 = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

Note that we have got an ideal point (point at infinity,  $\tilde{w} = 0$ ):

$$\tilde{l}_\infty \cdot (\tilde{l}_1 \times \tilde{l}_2) = (0 \ 0 \ 1)^T \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = 0$$

**The cross product for two parallel lines is therefore 0.**

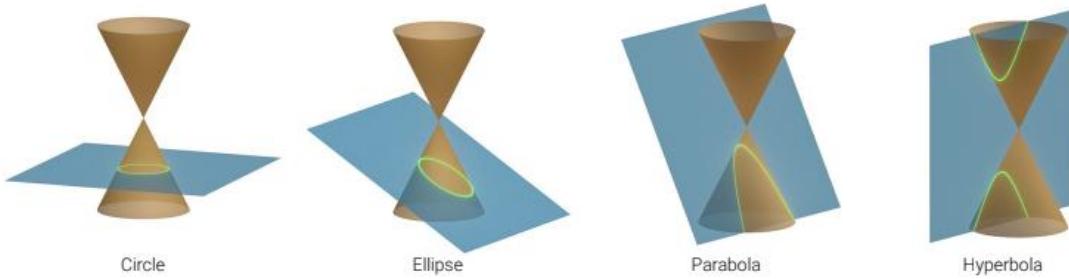
2D Conics

More complex algebraic objects can be represented using **polynomial homogeneous equations**.

For example, **conic sections** (arising as the intersection of a plane and a 3D cone) can be written using quadric equations:

$$\{\tilde{x} | \tilde{x}^T Q \tilde{x} = 0\}$$

The intersection of a plane (given by  $Q$ ) and a 3S cone can arise as parabola, circle, ellipse or hyperbola.



### Applying it to 3D

There is no such thing as a 3D projector (yet), so it's harder to imagine projective geometry in 3D, but the  $W$  value works exactly the same as it does in 2D. When  $W$  increases, the coordinate expands (scales up). When  $W$  decreases, the coordinate shrinks (scales down). The  $W$  is basically a scaling transformation for the 3D coordinate. Homogeneous vectors allow to express vectors at infinity, intersections of parallel lines and in general it allows to express transformations very easily as concatenations of multiple ones.

When  $W = 1$

The usual advice for 3D programming beginners is to always set  $W = 1$  whenever converting a 3D coordinate to a 4D coordinate. The reason for this is that when you scale a coordinate by 1 it doesn't shrink or grow, it just stays the same size. So, when  $W = 1$  it has no effect on the  $X$ ,  $Y$  or  $Z$  values.

For this reason, when it comes to 3D computer graphics, coordinates are said to be "correct" only when  $W = 1$ . If you rendered coordinates with  $W > 1$  then everything would look too **small**, and with  $W < 1$  everything would look too **big**. If you tried to render with  $W = 0$  your program would crash when it attempted to divide by zero. With  $W < 0$  everything would flip upside-down and back-to-front.

Mathematically speaking, there is no such thing as an "incorrect" homogeneous coordinate. Using coordinates with  $W = 1$  is just a useful convention for 3D computer graphics.

### 3D Planes

3D planes can also be represented as homogeneous coordinates  $\tilde{m} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$ :

$$\{\tilde{x} | \tilde{m}^T \tilde{x} = 0\} \Leftrightarrow \{x, y, z | ax + by + cz + d = 0\}$$

Again, we can normalize  $\tilde{m}$  so that  $\tilde{m} = \begin{pmatrix} n_x \\ n_y \\ n_z \\ d \end{pmatrix} = \begin{pmatrix} n \\ d \end{pmatrix}$  with  $\|n\|_2 = 1$ . In this case,  $n$  is the **normal perpendicular to the plane** and  $d$  is its **distance to the origin**.

An exception is the plane at infinity  $\tilde{m} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$  which **passes through all ideal points** (= points at infinity) for which  $\tilde{w} = 0$ .

### 3D Lines

**3D lines** are less elegant than either 2D lines or 3D planes. One possible representation is to express points on a line as a **linear combination** of two points  $p$  and  $q$  on the line:

$$\{x|x = (1 - \lambda)p + \lambda q\}, \quad \lambda \in \mathbb{R}$$

However, this representation uses 6 parameters for 4 degrees of freedom. Alternative minimal representations are the **two-plane parameterization** or **Plücker coordinates**.

## Direct Linear Transform

The **Direct Linear Transform (DLT)** is a general approach designed to solve systems of equations of the type:

$$\lambda x_k = A y_k; \quad \text{for } k = 1, 2, \dots, N$$

This type of equation frequently appears in projective geometry. One very important example is the relation between 3D points in a scene and their projection onto the image plane of a camera.

We want a homography estimation using a set of 2D correspondences - which means that we want to find the transformation matrix, as to say the camera parameters. In homography, we have 8 degrees of freedom, which means that we need at least 4 correspondence pairs for estimating the transformation matrix.

Let  $X = \{\tilde{x}_l^{(i)}, \tilde{x}_r^{(i)}\}_{i=1}^N$  denote a set of  $N$  2D-to-2D correspondences related by  $\tilde{x}_r = \tilde{H}\tilde{x}_l$ . As the correspondence vectors are homogeneous, they have the same direction but differ in magnitude. Thus, the equation above can be expressed as  $\tilde{x}_r^{(i)} \times \tilde{H}\tilde{x}_l^{(i)} = 0$ . We use this expression such that we assure that the vectors are equivalent (with respect to homogeneous coordinates). Using  $\tilde{h}_k^T$  to denote the  $k^{th}$  row of  $\tilde{H}$ , this can be rewritten as a linear equation in  $\tilde{h}$ :

$$\underbrace{\begin{bmatrix} 0^T & -\tilde{w}_r^{(i)}\tilde{x}_l^{(i)T} & \tilde{y}_r^{(i)}\tilde{x}_l^{(i)T} \\ \tilde{w}_r^{(i)}\tilde{x}_l^{(i)T} & 0^T & -\tilde{x}_r^{(i)}\tilde{x}_l^{(i)T} \\ -\tilde{y}_r^{(i)}\tilde{x}_l^{(i)T} & \tilde{x}_r^{(i)}\tilde{x}_l^{(i)T} & 0^T \end{bmatrix}}_{A_t} \begin{bmatrix} \tilde{h}_1 \\ \tilde{h}_2 \\ \tilde{h}_3 \end{bmatrix} = 0$$

Each point correspondence yields two equations. Stacking all equations into a  $2N \times 9$  dimensional matrix  $A$  leads to the following **constrained least squares problem**:

$$\tilde{h}^* = \arg \min_{\tilde{h}} \|A\tilde{h}\|_2^2 + \lambda (\|\tilde{h}\|_2^2 - 1) = \arg \min_{\tilde{h}} \tilde{h}^T A^T A \tilde{h} + \lambda (\tilde{h}^T \tilde{h} - 1)$$

where we have fixed  $\|\tilde{h}\|_2^2 = 1$  as  $\tilde{H}$  is homogeneous (i.e., defined only up to scale) and the trivial solution to  $\tilde{h} = 0$  is not of interest.

The solution to the above optimization problem is the **singular vector** corresponding to the smallest singular value of  $A$  (i.e., the last column of  $V$  when decomposing  $A = UDV^T$ ). The resulting algorithm is called **Direct Linear Transformation**.

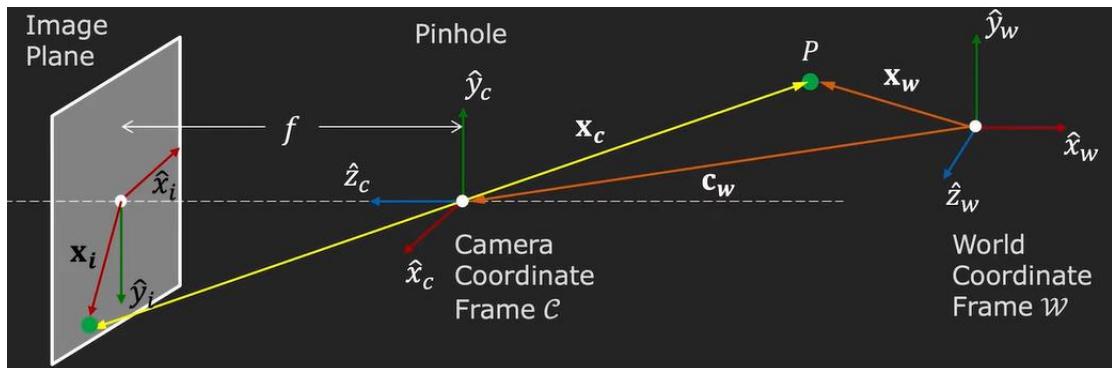
## Camera Calibration (Pose Estimation)

It is a method to find a camera's internal and external parameters. Where:

	Structure (scene geometry)	Motion (camera geometry)	<b>Measurements</b>
<b>Camera Calibration (a.k.a. Pose Estimation)</b>	known	estimate	3D to 2D correspondences
<b>Triangulation</b>	estimate	known	2D to 2D correspondences
<b>Reconstruction</b>	estimate	estimate	2D to 2D correspondences

## LINEAR CAMERA MODEL

The **Forward Imaging Model**, takes 3D points to 2D.



If we know the relative position of the camera coordinate frame with respect to the world coordinate frame, then we can write an expression that takes the point  $P$  from world coordinate frame to its projection on the image plane. This complete mapping is called **forward imaging model**.

The point  $P$  in the world coordinate frame (3D):

$$P = x_w = \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix}$$

We then model its transformation to the camera coordinate frame (3D):

$$x_c = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix}$$

Once we have those, we can apply **perspective projection** to end up with image coordinates (2D):

$$\boldsymbol{x}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

### Perspective Projection

We know that:

$$\begin{cases} \frac{x_i}{f} = \frac{x_c}{z_c} \\ \frac{y_i}{f} = \frac{y_c}{z_c} \end{cases} \Rightarrow \begin{cases} x_i = f \frac{x_c}{z_c} \\ y_i = f \frac{y_c}{z_c} \end{cases} \Rightarrow \tilde{\boldsymbol{x}}_i = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tilde{\boldsymbol{x}}_c$$

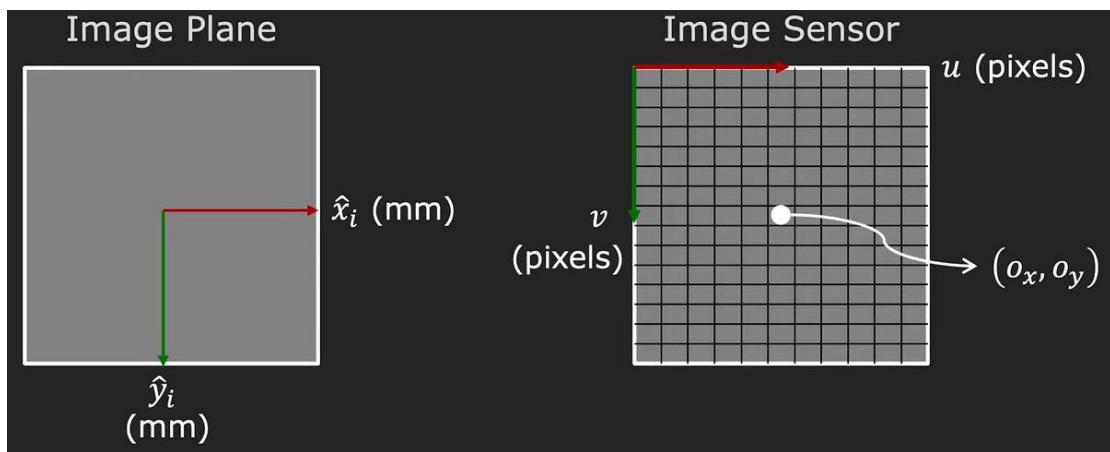
We know the points in [mm], but in reality we have an image sensor which is used to capture the image, that has **pixels**, those pixels may be rectangular.

If  $m_x$  and  $m_y$  are the pixel densities ( $\frac{\text{pixels}}{\text{mm}}$ ) in  $x$  and  $y$  directions, respectively, then pixel coordinates are:

$$\begin{cases} u = m_x x_i = m_x f \frac{x_c}{z_c} \\ v = m_y y_i = m_y f \frac{y_c}{z_c} \end{cases}$$

Here we have assumed that we know where the center of the image is located (where the optical axis pierces the image plain). We usually treat the top-left corner of the image sensor as its origin (easier for indexing). If pixel  $(o_x, o_y)$  is the **principal point** where the optical axis pierces the sensor, then:

$$\begin{cases} u = m_x f \frac{x_c}{z_c} + o_x \\ v = m_y f \frac{y_c}{z_c} + o_y \end{cases}$$



We can combine the unknown and get:

$$\begin{cases} u = f_x \frac{x_c}{z_c} + o_x \\ v = f_y \frac{y_c}{z_c} + o_y \end{cases}$$

Where:

$$\begin{cases} f_x = m_x f \\ f_y = m_y f \end{cases}$$

Are the focal lengths in pixels in the  $x$  and  $y$  directions.

$(f_x, f_y, o_x, o_y)$  are **Intrinsic Parameters** of the camera, they represent the **camera's internal geometry**.

The equations for perspective projection are Non-Linear  $(\frac{x_c}{z_c} \text{ or } \frac{y_c}{z_c})$ , it's convenient to express them as linear equations.

### *Homogeneous Coordinates*

The **homogeneous** representation of 2D point  $u = (u, v)^T$  is a 3D point  $\tilde{u} = (\tilde{u}, \tilde{v}, \tilde{w})^T$ . The third coordinate  $\tilde{w} \neq 0$  is fictitious such that:

$$\begin{cases} u = \frac{\tilde{u}}{\tilde{w}} \\ v = \frac{\tilde{v}}{\tilde{w}} \end{cases}$$

Hence:

$$u = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \tilde{w}u \\ \tilde{w}v \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \tilde{u}$$

The **homogeneous** representation of 3D point  $x = (x, y, z)^T \in \mathbb{R}^3$  is a 4D point  $\tilde{x} = (\tilde{x}, \tilde{y}, \tilde{z}, \tilde{w})^T \in \mathbb{R}^4$ . The fourth coordinate  $\tilde{w} \neq 0$  is fictitious such that:

$$\begin{cases} x = \frac{\tilde{x}}{\tilde{w}} \\ y = \frac{\tilde{y}}{\tilde{w}} \\ z = \frac{\tilde{z}}{\tilde{w}} \end{cases}$$

Hence:

$$x = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \tilde{w}x \\ \tilde{w}y \\ \tilde{w}z \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \\ \tilde{w} \end{bmatrix} = \tilde{x}$$

Using homogeneous coordinates of  $(u, v)$ :

$$u = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} z_c u \\ z_c v \\ z_c \end{bmatrix} = \begin{bmatrix} z_c \left( f_x \frac{x_c}{z_c} + o_x \right) \\ z_c \left( f_y \frac{y_c}{z_c} + o_y \right) \\ z_c \end{bmatrix} = \begin{bmatrix} f_x x_c + z_c o_x \\ f_y y_c + z_c o_y \\ z_c \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\text{Intrinsic Matrix}} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

*Intrinsic matrix*

$$M_{int} = [K|0] = \begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

We can decompose the camera matrix in the following way:

$$M_{int} = \begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The **intrinsic matrix** has all the internal parameters, where the sub-block  $K$  is the **Calibration Matrix**:

$$K = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix}$$

It is an **Upper-Right-Triangular Matrix**, it an important property that we will exploit later.



So:

$$\tilde{u} = [K|0]\tilde{x}_c = M_{int}\tilde{x}_c$$

Now, after we have mapped from camera coordinates to image coordinates, we need to map from the world coordinates to the camera coordinates.

But, before moving on, we will show the weak perspective camera model.

Weak Perspective Projection

If the relative distance  $\delta z_c$  (scene depth) between two points of a 3D object along the optical axis is much smaller than the average distance  $\bar{z}_c$  to the camera (example:  $\delta z_c < \frac{\bar{z}_c}{20}$ ), i.e.  $z_c \approx \bar{z}_c$ . Then:

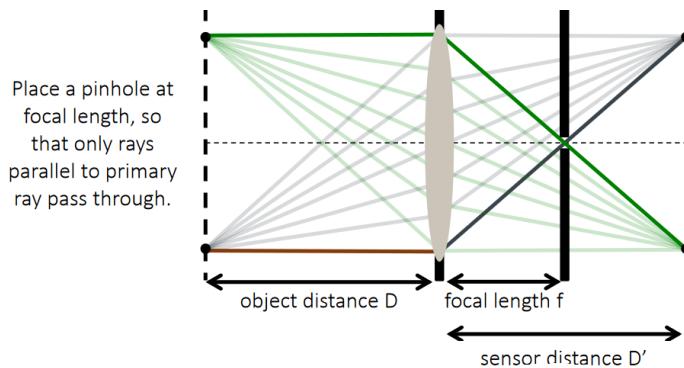
$$\begin{cases} u = f_x \frac{x_c}{z_c} + o_x \approx f_x \frac{x_c}{\bar{z}_c} + o_x \\ v = f_y \frac{y_c}{z_c} + o_y \approx f_y \frac{y_c}{\bar{z}_c} + o_y \end{cases}$$

We can assume a weak perspective camera in the following cases:

- When the scene (or part of it) is very far away.



- When we use a telecentric lens.



The affine camera becomes a **weak-perspective camera** when the rows of  $M_{int}$  form a uniformly scaled rotation matrix. The simplest form is (for  $f_x = f_y = f, o_x = o_y = 0$ ):

$$T_{wp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \bar{z}_c \\ 0 & 0 & 0 & f \end{bmatrix}$$

Yielding:

$$\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 0 & \bar{z}_c \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} \xrightarrow{\tilde{w}=\bar{z}_c} \begin{cases} u = \frac{\tilde{u}}{\tilde{w}} = \frac{fx_c}{\bar{z}_c} \\ v = \frac{\tilde{v}}{\tilde{w}} = \frac{fy_c}{\bar{z}_c} \end{cases}$$

The weak-perspective model is valid when the average variation of the depth of the object ( $\Delta z$ ) along the line of sight is small compared to the  $\bar{z}_c$  and the field of view is small. We see this as follows, expanding the perspective projection equation (for  $x$  coordinates, can show the same for  $y$ , or in vector form) using a Taylor series, we obtain:

$$u = \frac{f}{\bar{z}_c + \Delta z} x_c = \frac{f}{\bar{z}_c} \left( 1 - \frac{\Delta z}{\bar{z}_c} + \left( \frac{\Delta z}{\bar{z}_c} \right)^2 - \dots \right) x_c$$

When  $|\Delta z| \ll z_c$  only the zero-order term remains giving the weak-perspective projection. The error in image position is then  $x_{err} = x_p - x_{wp}$  (where  $x_p = \frac{f}{\bar{z}_c} x_c$ ):

$$x_{err} = \frac{f}{\bar{z}_c} x_c - \frac{f}{\bar{z}_c + \Delta z} x_c = \frac{f}{\bar{z}_c} \left( \frac{\Delta z}{\bar{z}_c + \Delta z} \right) x_c$$

showing that a small focal length  $f$ , small field of view  $\frac{x_c}{\bar{z}_c}$  and  $\frac{y_c}{\bar{z}_c}$  and small depth variation  $\Delta z$  contribute to the validity of the model.

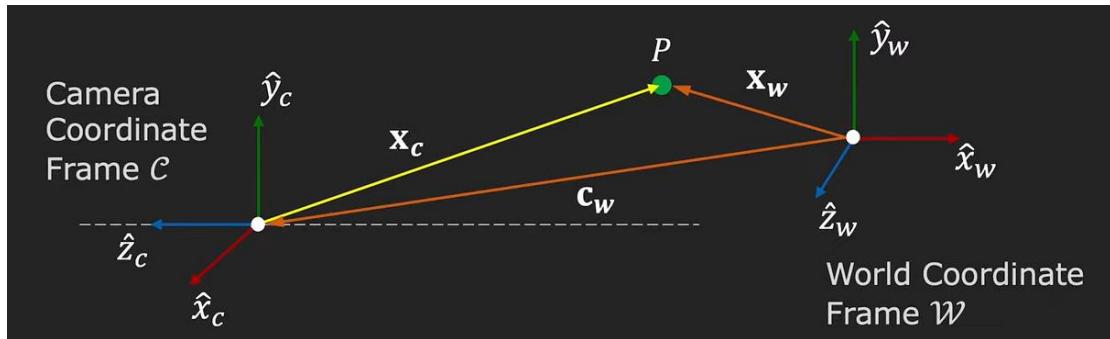
Special case of weak perspective camera is the orthographic camera, where:

- Constant magnification is equal to 1.
- There is no shift between camera and image origins.
- the world and camera coordinate systems are the same.

So:

$$M_{int} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### Extrinsic Parameters



**Position  $c_w$**  and **Orientation  $R$**  of the camera in the world coordinate frame  $w$  are the camera's **Extrinsic Parameters**.

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

$R$  is the orientation matrix where:

- The first row is the direction of  $\hat{x}_c$  in world coordinate frame
- The second row is the direction of  $\hat{y}_c$  in world coordinate frame
- The third row is the direction of  $\hat{z}_c$  in word coordinate frame

The matrix is an **orthonormal matrix**.

**Reminder:**

**Orthonormal Vectors:**

Two vectors  $u$  and  $v$  are orthonormal iff:

$$\text{dot}(u, v) = u \cdot v = u^T v = 0 \quad \text{and} \quad u^T u = v^T v = 1$$

**Orthonormal Matrix:**

A square matrix  $R$  whose row (or column) vectors are orthonormal. For such matrix:

$$R^{-1} = R^T \quad \text{and} \quad R^T R = R R^T = I$$

Given the **extrinsic parameters** ( $R, c_w$ ) of the camera, the camera-centric location of the point  $P$  in the world coordinate frame is:

$$x_c = R(x_w - c_w) = Rx_w - Rc_w = Rx_w + t$$

Where  $t$  is the **translation vector**:

$$t = -Rc_w$$

In matrix vector form:

$$\begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Rewriting using homogeneous coordinates:

$$\tilde{x}_c = \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

Where the **extrinsic matrix**:

$$M_{ext} = \begin{bmatrix} R_{3 \times 3} & t \\ 0_{1 \times 3} & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Hence:

$$\tilde{x}_c = M_{ext} \tilde{x}_w$$

So we have:

Camera to Pixel	World to Camera
$\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$	$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$
$\tilde{\mathbf{u}} = M_{int} \tilde{\mathbf{x}}_c$	$\tilde{\mathbf{x}}_c = M_{ext} \tilde{\mathbf{x}}_w$

Combining the above two equations, we get the full **projection matrix**  $P$ :

$$\tilde{\mathbf{u}} = M_{int} M_{ext} \tilde{\mathbf{x}}_w = P \tilde{\mathbf{x}}_w$$

From the projection matrix we sometimes (like in the tutorial) we show the camera matrix as:

$$M = K[R|t]$$

Where:

- $M$  is  $3 \times 4$  matrix
- $K$  is the calibration matrix ( $3 \times 3$  matrix)

$$K = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix}$$

- $R$  is the orientation matrix ( $3 \times 3$  matrix)

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

- $t$  is the translation vector ( $3 \times 1$  vector)

$$t = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}$$

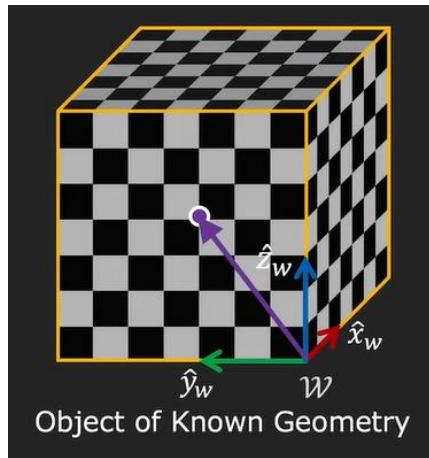
Now we have a linear camera model, which is given by the projection matrix, so we are ready to calibrate the camera, which essentially means to estimate the projection matrix.

$$\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

## CAMERA CALIBRATION (GEOMETRIC)

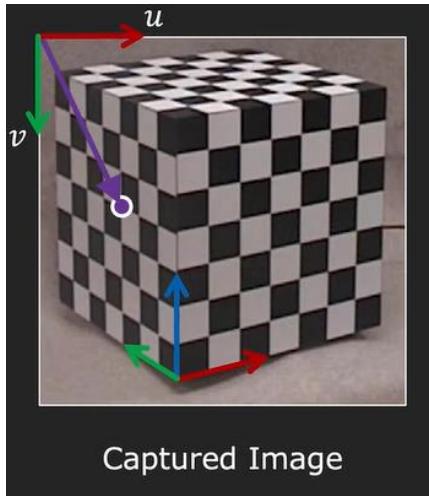
We are ready now to **calibrate the camera**, which essentially means **estimate the projection matrix**.

At the **first step**, we will use an object of **known geometry**, like this one:



We can place our world coordinate frame on one of the corners of this cube.

We take a single image of this object (cube):



The **second step** is to identify correspondences between 3D scene points and image points.

If we consider a single point -  $x_w$  in the world coordinate frame, and we know its image location -  $u$  (we also do the same for all visual features, can be done manually or with some algorithm, as it is a simple object).

The **third step** is for each corresponding point  $i$  in scene and image:

$$\begin{bmatrix} u^{(i)} \\ v^{(i)} \\ 1 \end{bmatrix}_{\text{Known}} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix}_{\text{Unknown}} \begin{bmatrix} x_w^{(i)} \\ y_w^{(i)} \\ z_w^{(i)} \\ 1 \end{bmatrix}_{\text{Known}}$$

Expanding the matrix as linear equations:

$$\begin{cases} u^{(i)} = \frac{p_{11}x_w^{(i)} + p_{12}y_w^{(i)} + p_{13}z_w^{(i)} + p_{14}}{p_{31}x_w^{(i)} + p_{32}y_w^{(i)} + p_{33}z_w^{(i)} + p_{34}} \\ v^{(i)} = \frac{p_{21}x_w^{(i)} + p_{22}y_w^{(i)} + p_{23}z_w^{(i)} + p_{24}}{p_{31}x_w^{(i)} + p_{32}y_w^{(i)} + p_{33}z_w^{(i)} + p_{34}} \end{cases}$$

At the **fourth step**, we are rearranging the terms:

$$\begin{bmatrix} x_w^{(1)} & y_w^{(1)} & z_w^{(1)} & 1 & 0 & 0 & 0 & -u_1x_w^{(1)} & -u_1y_w^{(1)} & -u_1z_w^{(1)} & -u_1 \\ 0 & 0 & 0 & 0 & x_w^{(1)} & y_w^{(1)} & z_w^{(1)} & 1 & -v_1x_w^{(1)} & -v_1y_w^{(1)} & -v_1z_w^{(1)} & -v_1 \\ \vdots & \vdots \\ x_w^{(i)} & y_w^{(i)} & z_w^{(i)} & 1 & 0 & 0 & 0 & -u_ix_w^{(i)} & -u_iy_w^{(i)} & -u_iz_w^{(i)} & -u_i \\ 0 & 0 & 0 & 0 & x_w^{(i)} & y_w^{(i)} & z_w^{(i)} & 1 & -v_ix_w^{(i)} & -v_iy_w^{(i)} & -v_iz_w^{(i)} & -v_i \\ \vdots & \vdots \\ x_w^{(n)} & y_w^{(n)} & z_w^{(n)} & 1 & 0 & 0 & 0 & -u_nx_w^{(n)} & -u_ny_w^{(n)} & -u_nz_w^{(n)} & -u_n \\ 0 & 0 & 0 & 0 & x_w^{(n)} & y_w^{(n)} & z_w^{(n)} & 1 & -v_nx_w^{(n)} & -v_ny_w^{(n)} & -v_nz_w^{(n)} & -v_n \end{bmatrix} = \begin{bmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{14} \\ p_{21} \\ p_{22} \\ p_{23} \\ p_{24} \\ p_{31} \\ p_{32} \\ p_{33} \\ p_{34} \end{bmatrix}$$

Or as:

$$Ap = 0$$

At the fifth step we solve for  $p$ :

$$Ap = 0$$

We know that projection matrix  $P$  and  $kP$  (scaled) produce the same homogeneous pixel coordinates. Which means that, **Projection Matrix  $P$  is determined up to a scale**.

Scaling projection matrix implies simultaneously scaling the world and camera, which does not change the image. So we can **set projection matrix to some arbitrary scale**.

One option is to set  $p_{ij} = 1$ . Or more cleverly, set the scale so:

$$\|p\|^2 = 1$$

And we can formulate our problem, as we want  $Ap$  as close to 0 as possible and  $\|p\|^2 = 1$ :

$$\min_p \|Ap\|^2 \text{ such that } \|p\|^2 = 1$$

Or:

$$\min_p (p^T A^T A p) \text{ such that } p^T p = 1$$

Define **Loss function**  $L(p, \lambda)$ :

$$L(p, \lambda) = p^T A^T A p - \lambda(p^T p - 1)$$

Taking derivatives of  $L(p, \lambda)$  with respect to  $p$ :

$$2A^TAp - 2\lambda p = 0$$

It is equivalent to solving the eigenvalue problem:

$$A^TAp = \lambda p$$

Where, eigenvector  $p$  with **smallest eigenvalue**  $\lambda$  of matrix  $A^TA$  minimizes the loss  $L(p)$ .

At the final step we rearrange solution  $p$  to form the projection matrix  $P$ .

### INTRINSIC AND EXTRINSIC MATRICES

Using our calibration method, we know how to estimate the projection matrix. We can take the projection matrix and decompose it into the intrinsic matrix (which has all the internal parameters) and the extrinsic matrix (which has the external parameters of the camera).

We know that:

$$P = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{M_{int} = \text{Intrinsic Matrix}} \underbrace{\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{M_{ext} = \text{Extrinsic Matrix}}$$

Where:

$$\begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{pmatrix} = \begin{pmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} = KR$$

Given that  $K$  is an **Upper-Right-Triangular** matrix and  $R$  is an **Orthonormal** matrix, it is possible to uniquely “**decouple**”  $K$  and  $R$  from their product using “**QR factorization**”.

We can also find the **translation**, where:

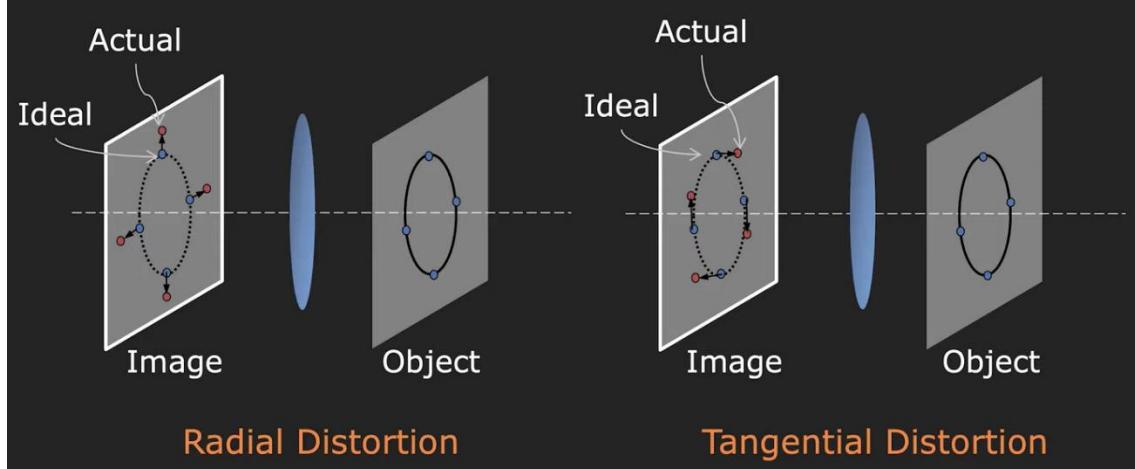
$$\begin{bmatrix} p_{14} \\ p_{24} \\ p_{34} \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} = Kt$$

We have found  $K$ , therefore:

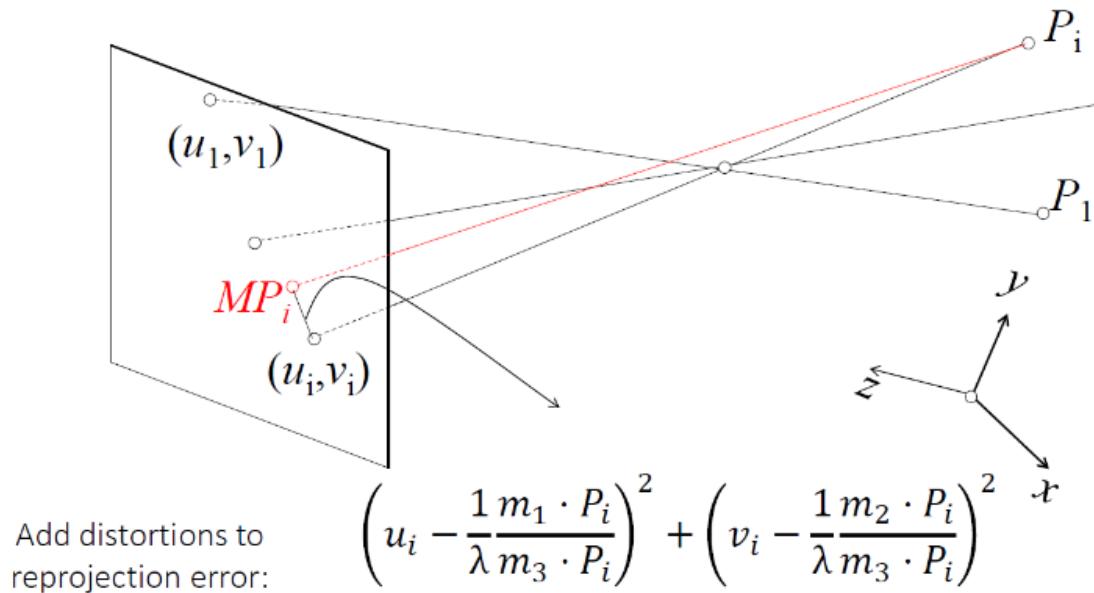
$$t = K^{-1} \begin{bmatrix} p_{14} \\ p_{24} \\ p_{34} \end{bmatrix}$$

*Other Intrinsic Parameters*

Pinholes do not exhibit image distortions. But, lens do!



The intrinsic model of the camera will need to include the distortion coefficients.

Where the **radial distortion** is:

$$\begin{cases} x_{rad} = x[1 + k_1 r^2 + k_2 r^4 + k_3 r^6] \\ y_{rad} = y[1 + k_1 r^2 + k_2 r^4 + k_3 r^6] \end{cases}$$

And the **tangential distortion**:

$$\begin{cases} x_{tan} = x + [2p_1 xy + p_2(r^2 + 2x^2)] \\ x_{tan} = x + [p_1(r^2 + 2y^2) + 2p_2 xy] \end{cases}$$

We end up with 5 parameters to estimate:

$$\text{distortion coefficients} = [k_1, k_2, k_3, p_1, p_2]^T$$

## GEOMETRIC CAMERA CALIBRATION PROS AND CONS

### **Advantages:**

- Very simple to formulate
- Analytical solution

### **Disadvantages:**

- Doesn't model radial distortion
- Hard to impose constraints
- Doesn't minimize the correct error function

For these reasons, nonlinear methods are preferred:

- Define error function  $E$  between projected 3D points and image positions
  - $E$  is nonlinear function of intrinsic, extrinsic and radial distortion
- Minimize  $E$  using nonlinear optimization techniques

### **Alternative – Multi-plane calibration**

### **Advantages:**

- Only requires a plane
- Don't have to know positions/orientations

### **Disadvantages:**

- Need to solve non-linear optimization problem

## HOMOGRAPHY PROOFS (EXAMPLES)

### 1. Homography of planar object

Prove that a  $3 \times 3$  homography transform  $H$  is sufficient to describe the mapping between a **planar 3D object** and a camera, i.e. point matches of the form  $\{p_i, P_i\}$ , where  $p_i = [x_i, y_i, w_i]^T$  and  $P_i = [X_i, Y_i, Z_i, 1]^T$  satisfying  $aX_i + bY_i + cZ_i + d = 0$

#### Proof:

In this special case of a **planar scene**, we do not need the full  $3 \times 4$  camera matrix  $M$  and we can make due with a  $3 \times 3$  **homography matrix**  $H$ . The proof is relatively straight forward, and rely on the following observation:

Since the points in 3D lie on a plane (it satisfies the plane equation):

$$aX + bY + cZ + d = 0$$

Hence:

$$aX_i + bY_i + cZ_i + d = 0 \Rightarrow Z_i = \frac{-aX_i - bY_i - d}{c}$$

And write the point  $P_i$  as:

$$P_i = \left[ X_i, Y_i, -\frac{aX_i}{c} - \frac{bY_i}{c} - \frac{d}{c}, 1 \right]$$

This leads to the main conclusion that, the **4D homogeneous coordinates are redundant** and can be written down by a 3D homogeneous coordinates:

$$\begin{bmatrix} X_i \\ Y_i \\ Z_i \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{a}{c} & -\frac{b}{c} & -\frac{d}{c} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ 1 \end{bmatrix}$$

Now coming back to the general camera matrix how can we conclude it can be reduced to a homography? The answer lies in simplifying the matrix-vector product. A general matrix  $M$  satisfy the relation:

$$\begin{bmatrix} x_i \\ y_i \\ w_i \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{bmatrix}$$

Assuming the scene is planar, we plug in the plane equation for  $Z_i = \frac{-aX_i - bY_i - d}{c}$ :

$$\begin{bmatrix} x_i \\ y_i \\ w_i \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ -\frac{aX_i}{c} - \frac{bY_i}{c} - \frac{d}{c} \\ 1 \end{bmatrix}$$

Now, let us look more closely at the formula for  $x_i$ :

$$\begin{aligned}x_i &= m_{11}X_i + m_{12}Y_i + m_{13}\left(-\frac{aX_i}{c} - \frac{bY_i}{c} - \frac{d}{c}\right) + m_{14} = \\&= \left(m_{11} - \frac{a}{c}m_{13}\right)X_i + \left(m_{12} - \frac{b}{c}m_{13}\right)Y_i + \left(m_{14} - \frac{d}{c}m_{13}\right)\end{aligned}$$

Similarly, this can be done for  $y_i$  and  $w_i$ :

$$\begin{aligned}y_i &= \left(m_{21} - \frac{a}{c}m_{23}\right)X_i + \left(m_{22} - \frac{b}{c}m_{23}\right)Y_i + \left(m_{24} - \frac{d}{c}m_{23}\right) \\w_i &= \left(m_{31} - \frac{a}{c}m_{33}\right)X_i + \left(m_{32} - \frac{b}{c}m_{33}\right)Y_i + \left(m_{34} - \frac{d}{c}m_{33}\right)\end{aligned}$$

Therefore, rewriting this in matrix form we get the following relation:

$$\begin{bmatrix}x_i \\ y_i \\ w_i\end{bmatrix} = \begin{bmatrix}m_{11} - \frac{a}{c}m_{13} & m_{12} - \frac{b}{c}m_{13} & m_{14} - \frac{d}{c}m_{13} \\ m_{21} - \frac{a}{c}m_{23} & m_{22} - \frac{b}{c}m_{23} & m_{24} - \frac{d}{c}m_{23} \\ m_{31} - \frac{a}{c}m_{33} & m_{32} - \frac{b}{c}m_{33} & m_{34} - \frac{d}{c}m_{33}\end{bmatrix} \begin{bmatrix}X_i \\ Y_i \\ 1\end{bmatrix}$$

Meaning, the 3D point  $P_i$  can be indeed reduced to a 3D homogeneous  $[X_i, Y_i, 1]^T$  vector, and is related to the 2D point in the image  $p_i = [x_i, y_i, w_i]^T$  through a  $3 \times 3$  homography  $H$  that is a function of the entries in the general  $3 \times 4$  camera matrix  $M$ , and the normal to the plane  $[a, b, c, d]^T$ .

## 2. Homography between 2 points

Prove that there exists a homography  $H$  that satisfies:

$$p_1 = Hp_2$$

between the 2D points (in homogeneous coordinates)  $p_1$  and  $p_2$  in the images of a plane  $\Pi$  captured by two  $3 \times 4$  camera projection matrices  $M_1$  and  $M_2$ , respectively. Where the equality is up to a scale.

**Note:** A degenerate case happens when the plane  $\Pi$  contains both cameras' centers, in which case there are infinite choices of  $H$  satisfying the equation. You can ignore this special case in your answer.

**Proof:**

Plane in 3D using homogeneous coordinates is given by:

$$n^T P = 0$$

Where  $n$  is the normal to the plane and  $P$  is a homogeneous vector, both are  $4 \times 1$  vectors. Therefore, we can find a basis of 3 vectors  $\{u_1, u_2, u_3\}$  in  $\mathbb{R}^4$ , such that each point on the plane is given by:

$$P = \sum_{i=1}^3 \alpha_i u_i$$

The projection of 3D point  $P$  to the  $j^{th}$  image point  $p_j$  ( $p_1 = M_1 P$  and  $p_2 = M_2 P$ ) is given by:

$$p_j = M_j P = \sum_{i=1}^3 \alpha_i M_j u_i$$

If we denote  $v_j^i = M_j u_i$  we get:

$$\begin{aligned} p_1 &= \sum_{i=1}^3 \alpha_i v_1^i \\ p_2 &= \sum_{i=1}^3 \alpha_i v_2^i \end{aligned}$$

Hence, the relation between the two points is a  $3 \times 3$  matrix satisfying:

$$\begin{bmatrix} | & | & | \\ v_1^1 & v_1^2 & v_1^3 \\ | & | & | \end{bmatrix} = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \begin{bmatrix} | & | & | \\ v_2^1 & v_2^2 & v_2^3 \\ | & | & | \end{bmatrix}$$

Relation to camera calibration:

- Recall that we have 11 degrees of freedom in  $M$ .
- If all the calibration points are on a plane, we get at most 8 independent equations out of 4 points.
- Any additional point will result in constraints that are linearly dependent on the constraints from the previous 4 points on the plane.

Therefore, **in estimating  $M$  we can't rely on a single image of the chessboard.**

### 3. Pure rotation

Prove that there exists a homography  $H$  that satisfies the equation  $p_1 = H p_2$ , given two cameras separated by a pure rotation. That is, for camera 1,  $p_1 = K_1[I|0]P$ , and for camera 2,  $p_2 = K_2[R|0]P$ . Note that  $K_1$  and  $K_2$  are the intrinsic matrices of the cameras and are different.  $I$  is  $3 \times 3$  identity matrix,  $0$  is a zero vector and  $P$  is a point in 3D space (in homogeneous coordinates).  $R$  is the rotation matrix of the camera.

Since the last column is zero, we can write:

$$\underbrace{p_2}_{3 \times 1} = \underbrace{K_2}_{3 \times 3} \underbrace{R}_{3 \times 3} \underbrace{P}_{3 \times 1} \Rightarrow P = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = R^{-1} K_2^{-1} p_2$$

Substituting this in the equation for  $p_1$  we get:

$$p_1 = K_1 I P = K_1 I R^{-1} K_2^{-1} p_2 = K_1 R^{-1} K_2^{-1} p_2$$

Where we can write the homography matrix as:

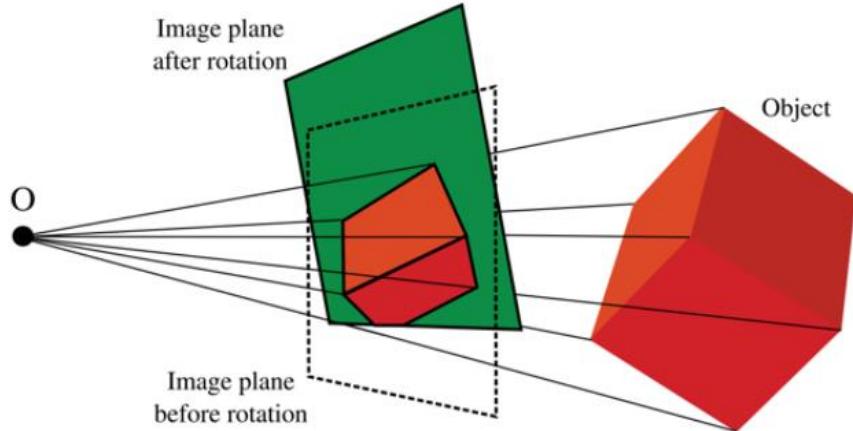
$$H = K_1 R^{-1} K_2^{-1}$$

**Conclusion:**

- If 2 cameras differ only in rotation, then we can't triangulate!

Remember where this was useful?

- Panorama stitching! (There we did not care about recovering depth)



**Figure 15.14** Images under pure camera rotation. When the camera rotates but does not translate, the bundle of rays remains the same, but is cut by a different plane. It follows that the two images are related by a homography.

#### 4. Angular rotation

Suppose that a camera is rotating about its center  $C$ , keeping the intrinsic parameters  $K$  constant. Let  $H$  be the homography that maps the view from one camera orientation to the view at a second orientation. Let  $\theta$  be the angle of rotation between the two. Show that  $H^2$  is the homography corresponding to a rotation of  $2\theta$ .

**Proof:**

We have just shown that for such a scenario:

$$\begin{aligned} H_{2 \rightarrow 1} &= K_1 R_\theta^{-1} K_2^{-1} \\ H_{1 \rightarrow 2} &= K_2 R_\theta K_1^{-1} \end{aligned}$$

Applying the constraint  $K_1 = K_2 = K$  gets us:

$$H_{1 \rightarrow 2} = K_2 R_\theta K_1^{-1} = K R_\theta K^{-1}$$

Applying  $H_{1 \rightarrow 2}$  twice gets us:

$$H_{1 \rightarrow 2}^2 = (H_{1 \rightarrow 2})(H_{1 \rightarrow 2}) = (K R_\theta K^{-1})(K R_\theta K^{-1}) = K R_\theta K^{-1} K R_\theta K^{-1} = K R_{2\theta} K^{-1}$$

Since  $R_\theta R_\theta = R_{2\theta}$ .

Which is a homography that corresponds to a rotation of  $2\theta$ .

### 5. The need for different directions

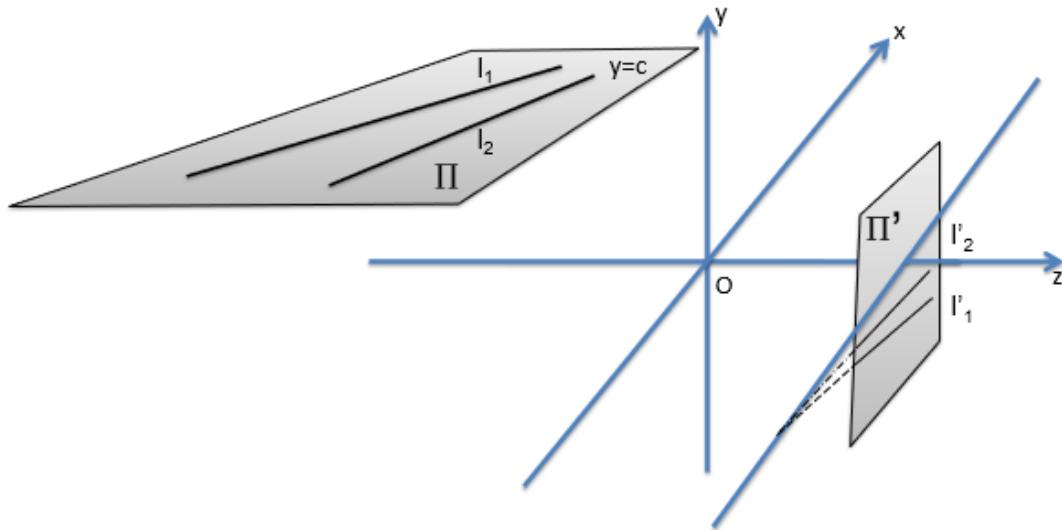
Prove that points on a single line do not uniquely constrain the homography  $H$ . In other words, prove that we need points on at least 2 different directions within the plane to estimate a homography reliably.

#### Proof:

If all points lie on a line there is a  $3 \times 1$  vector  $l$  such that the  $l^T p = 0$  for all points  $p$ . Now suppose you found a homography matrix  $H$  such that  $p'_j = Hp_j$ , and yet all your points satisfy  $l^T p_j = 0$ . Then it is easy to see that for every  $3 \times 1$  vector  $v$  the matrix  $H' = H + v l^T$  will also satisfy  $p'_j = H' p_j$ . This implies that there is no unique  $H$  which explains points on the same line.

### 6. Perspective Projection

In the following figure, there are two parallel lines  $l_1$  and  $l_2$  lying on the same plane  $\Pi$ .  $l'_1$  and  $l'_2$  are their projections through the optical center  $O$  on the image plane  $\Pi'$ . Let's define plane  $\Pi$  by  $y = c$ , line  $l_1$  by equation  $ax + bz = d_1$ , and line  $l_2$  by equation  $ax + bz = d_2$ .



- a. For any point  $P(x, y)$  on  $l_1$  or  $l_2$ , use the perspective projection equation below to find the projected point  $P' = (x', y')$  on the image plane.  $f'$  is the focal length of the camera. Express your answer in terms of  $a, b, c, d, z$  and  $f'$ .

$$\begin{cases} x' = f' \frac{x}{z} \\ y' = f' \frac{y}{z} \end{cases}$$

- b. It turns out  $l'_1$  and  $l'_2$  appear to converge on the intersection of the image plane  $\Pi'$  given by  $z = f'$  and the plane  $y = 0$ . Explain why.

#### Solution:

- a. According to the perspective projection equation, a point on  $l$  projects onto the image point defined by:

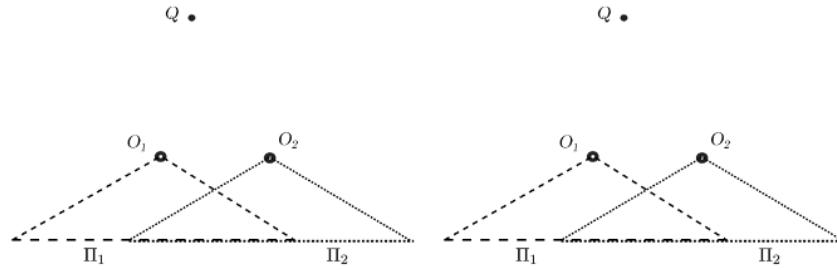
$$\begin{cases} x' = f' \frac{x}{z} = f' \frac{d - bz}{az} \\ y' = f' \frac{y}{z} = f' \frac{c}{z} \end{cases}$$

- b. This is a parametric representation of the image  $\delta$  of the line  $\Delta$  with  $z$  as the parameter.

This image is in fact only a half-line since when  $z \rightarrow \infty$ , it stops at the point  $(x', y') = \left(-f' \frac{b}{a}, 0\right)$  on the  $x'$  axis of the image plane. This is the vanishing point associated with all parallel lines with slope  $-\frac{b}{a}$  in the plane  $\Pi$ . All vanishing points lie on the  $x'$  axis, which is the horizon line in this case.

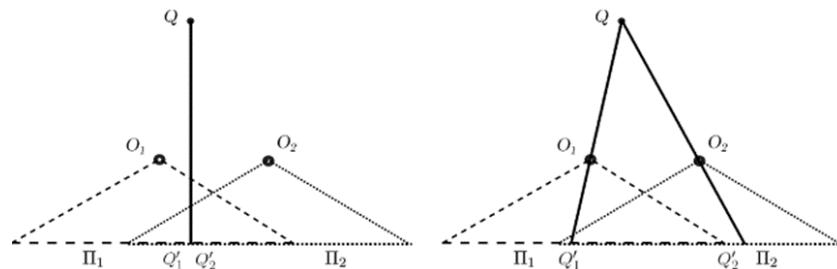
### 7. Point Reconstruction

As shown in the following figure, a point  $Q$  is observed in a known (i.e. intrinsic parameters are calibrated) **affine camera** with image plane  $\Pi_1$ . Then you translate the camera parallel to the image plane with a known translation to a new image plane  $\Pi_2$  and observe it again.



- Draw the image points  $Q'_1$  and  $Q'_2$  on  $\Pi_1$  and  $\Pi_2$ . Is it possible to find the depth of the 3D point  $Q$  in this scenario? Briefly explain why.
- What if this is a **perspective camera**? Draw  $Q'_1$  and  $Q'_2$ . Is it possible to find the depth of the 3D point  $Q$  in this scenario? Briefly explain why.

**Solution:**

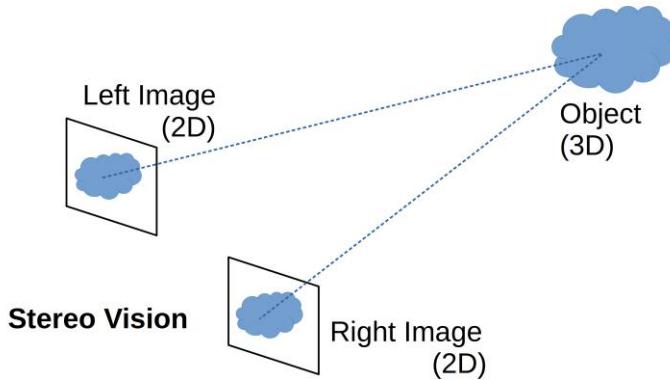


- Drawn on the left. No, we cannot determine point  $Q$  because it can be any 3D point on the line.
- Drawn on the right. Yes, because we can do triangulation in this case.

## Stereo 3D

**Computer stereo vision is the extraction of 3D information from 2D images, such as those produced by a CCD camera.** It compares data from multiple perspectives and combines the relative positions of things in each view. As such, we use stereo vision in applications like advanced driver assistance systems and robot navigation.

It's similar to how human vision works. Our brains' simultaneous integration of the images from both of our eyes results in 3D vision:



**Although each eye produces only a two-dimensional image, the human brain can perceive depth when combining both views and recognizing their differences.** We call this ability **stereo vision**.

In **Two-View Stereo Matching**, the **goal** is recovering the disparity for every pixel from the input images. The disparity (or inverse depth) is defined as the relative displacement between pixels in the two images.

Our **task** is to construct a dense 3D model from two images of a static scene, e.g. a scene captured with synchronized cameras.

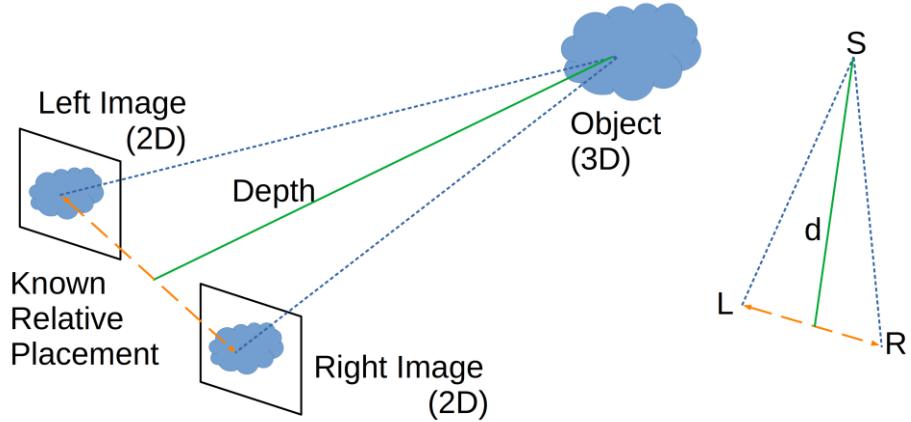
The **pipeline**:

1. **Calibrate cameras** intrinsically and extrinsically
2. **Rectify images** given the calibration
3. **Compute disparity map** for reference image
4. **Remove outliers** using consistency/occlusion test
5. **Obtain depth** from disparity using camera calibration
6. **Construct 3D model**, e.g. via volumetric fusion and meshing

Given a set of input images, we first compute the camera poses, e.g. using incremental bundle adjustment. The camera poses enable us to compute dense correspondences for each adjacent view via epipolar geometry. Correspondences can be used to recover depth maps. Using depth map fusion, these maps lead to a coherent 3D model that takes all made observations into account.

## RECEIVING DEPTH

Let's suppose there are left and right cameras, both producing a 2D image of a scene. Let  $S$  be a point on a real-world (3D) object in the scene:



To determine the depth of  $S$  in the composite 3D image, we first find two pixels  $L$  and  $R$  in the left and right 2D images that correspond to it. We can assume that we know the relative positioning of the two cameras. The computing system estimates the depth  $d$  by **triangulation** using the prior knowledge of the relative distance between the cameras.

For the left image (in pixel coordinates, from the left camera), the point is given by (3D to 2D):

$$\begin{cases} u_l = f_x \frac{x}{z} + o_x \\ v_l = f_y \frac{y}{z} + o_y \end{cases}$$

We also can show the 2D to 3D (the point lies on a ray):

$$\begin{cases} x = \frac{z}{f_x} (u - o_x) \\ y = \frac{z}{f_y} (v - o_y), \quad z > 0 \end{cases}$$

And for the right image (in pixel coordinates, from the right camera), the point is given by (it's a problem by itself to find those corresponding points – called **The Correspondence Problem**):

$$\begin{cases} u_r = f_x \frac{x - b}{z} + o_x \\ v_r = f_y \frac{y}{z} + o_y \end{cases}$$

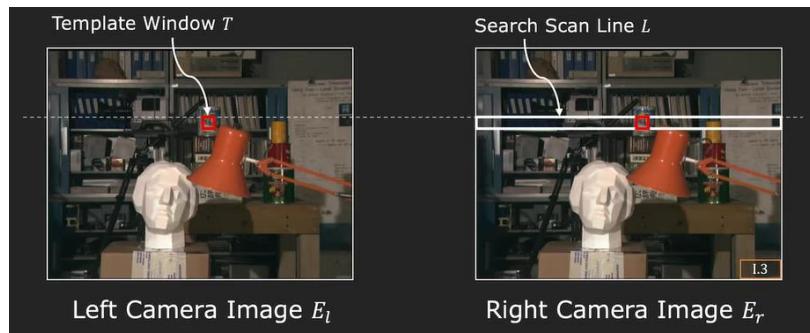
Where  $f_x, f_y, b, o_x, o_y$  are known ( $b$  is the known **horizontal baseline** between the left and the right images).

Solving for  $(x, y, z)$ :

$$\begin{cases} x = \frac{b(u_l - o_x)}{u_l - u_r} \\ y = \frac{bf_x(v_l - o_y)}{f_y(u_l - u_r)} \\ z = \frac{bf_x}{u_l - u_r} \end{cases}$$

Where  $u_l - u_r$  is the **Disparity**. Depth  $z$  is inversely proportional to the disparity, and the disparity is proportional to the baseline  $b$  of the system.

We can see that from perspective projection:  $v_l = v_r = f_y \frac{y}{z} + o_y$ . Which mean that the corresponding scene points lie on the **same horizontal scan line**. So to determine the disparity we can use **template matching**:



$$\text{So - Disparity: } d = u_l - u_r, \text{ Depth: } z = \frac{bf_x}{u_l - u_r}$$

There are some metrics for template matching, like:

- **Sum of Absolute Differences (SAD):**

$$SAD(k, l) = \sum_{(i,j) \in T} |E_l(i, j) - E_r(i + k, j + l)|$$

- **Minimum Sum of Square Differences (SSD):**

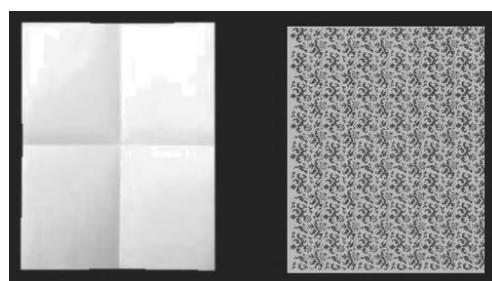
$$SSD(k, l) = \sum_{(i,j) \in T} |E_l(i, j) - E_r(i + k, j + l)|^2$$

- **Maximum Normalized Cross-Correlation:**

$$NCC(k, l) = \frac{\sum_{(i,j) \in T} E_l(i, j) E_r(i + k, j + l)}{\sqrt{\sum_{(i,j) \in T} E_l(i, j)^2 \sum_{(i,j) \in T} E_r(i + k, j + l)^2}}$$

### *Issues with Stereo Matching*

- Surface must have (non-repetitive) texture.



- Foreshortening effect makes matching challenging.

For better results we can use **Adaptive Window Method**. For each point, match using windows of multiple sizes and use the disparity that is a result of the best similarity measure (minimize SSD per pixel.)

The human brain works the same way. Its ability to perceive depth and 3D forms is known as stereopsis.

## SIAMESE NETWORKS

Hand crafted metrics do not take into consideration relevant geometric and radiometric invariances or occlusion patterns. Unfortunately, the world is too complex to specify this by hand. Instead, matching cost computation can be treated as a classification problem where pairs of patches from both images are labeled as "good match" or "bad match".

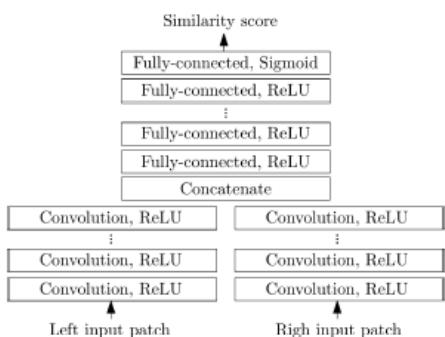
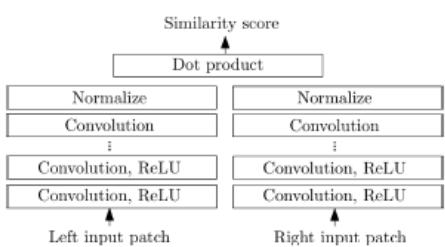
Left patch	Right patch	Label
		Good match
		Bad match

### Method Overview:

1. Train CNN patch-wise based on images with ground truth disparity maps.
2. Calculate features for each of the two images using learned model.
3. Correlate features between both images (dot product).
4. Find maximum for every pixel (winner-takes-all) or run global optimization algorithm that incorporates some smoothness assumptions.

### *Siamese Network Architecture*

The next figure shows two different architectures for the Siamese Network model. Both use convolutional layers to extract features from the two input images. The convolutions applied to the left and the right image share their parameters. The **Learned Similarity** model concatenates those features and applies several fully-connected layers (Multilayer Perceptron) to compute a similarity score. On the other hand, the **Cosine Similarity** model just normalizes the output of the convolution blocks and computes the dot product as similarity score. While the former approach has much larger computational cost, their performance is roughly on par.

Learned Similarity	Cosine Similarity
Learns features and the similarity metric.	Learns features and applies dot product.
Potentially more expressive.	Features must do the heavy lifting.
Slow ( $W \times H \times D$ MLP evaluations)	Fast matching (no network evaluation)
	

### Training

The training set is composed of patch triplets:

$$(w_L(x_L^{ref}), w_R(x_R^{neg}), w_R(x_R^{pos}))$$

where  $w_L(x_L)$  is an image patch from the left image centered at  $x_L = (x_L, y_L)$  and  $w_R(x_R)$  is an image patch from the right image centered at  $x_R = (x_R, y_R)$ . Negative and positive examples are created by **applying different offsets to true correspondence**. For the **negative examples**:

$$x_R^{neg} = (x_L^{ref} - d + o_{neg}, y_L^{ref})$$

the reference pixel is shifted by the ground truth disparity  $d$  and an offset  $o_{neg}$  drawn from:

$$\text{Uni}(\{-N_{hi}, \dots, -N_{lo}, N_{lo}, \dots, N_{hi}\})$$

For the positive examples:

$$x_R^{pos} = (x_L^{ref} - d + o_{pos}, y_L^{ref})$$

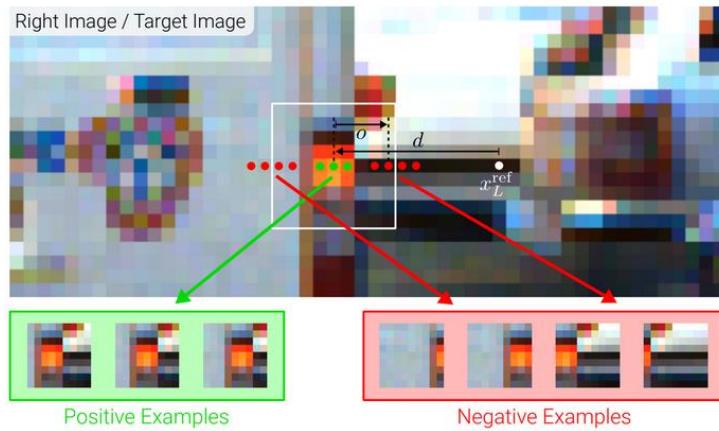
the offset  $o_{pos}$  drawn from a smaller range:

$$\text{Uni}(\{-P_{hi}, \dots, P_{hi}\})$$

This training process is called **hard negative mining** and, typically, the hyperparameters are chosen as:

$$P_{hi} = 1, N_{lo} = 3, N_{hi} = 6$$

Note that the negative examples are chosen very close to the positive examples, but far enough for the classifier to classify them correctly.



### Loss Function

The loss function used for the training process is the **Hinge Loss**:

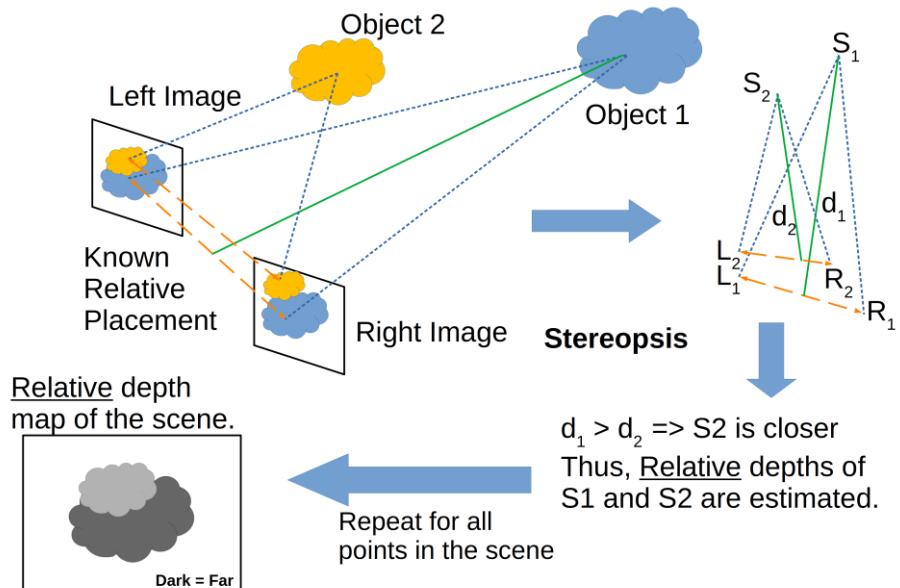
$$L = \max(0, m + s_- - s_+)$$

Here,  $s_-$  and  $s_+$  are the scores of the network for the negative and positive example respectively. Therefore, the loss is zero when the similarity of the positive example is greater than the similarity of the negative example by at least margin  $m$  (tunable hyperparameter). This prevents the further separation of positive and negative features that are already well separated and gives the model the capacity to focus on the hard cases.

### HOW COMPUTER SYSTEMS ACHIEVE STEREO VISION

We need to estimate each point's depth to produce a 3D image from two-dimensional ones.

From there, we can determine the points' relative depths and get a depth map:

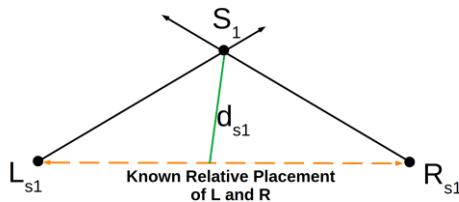


A depth map is an image (or image channel) that contains the data on the separation between the surfaces of scene objects from a viewpoint. This is a common way to represent scene depths

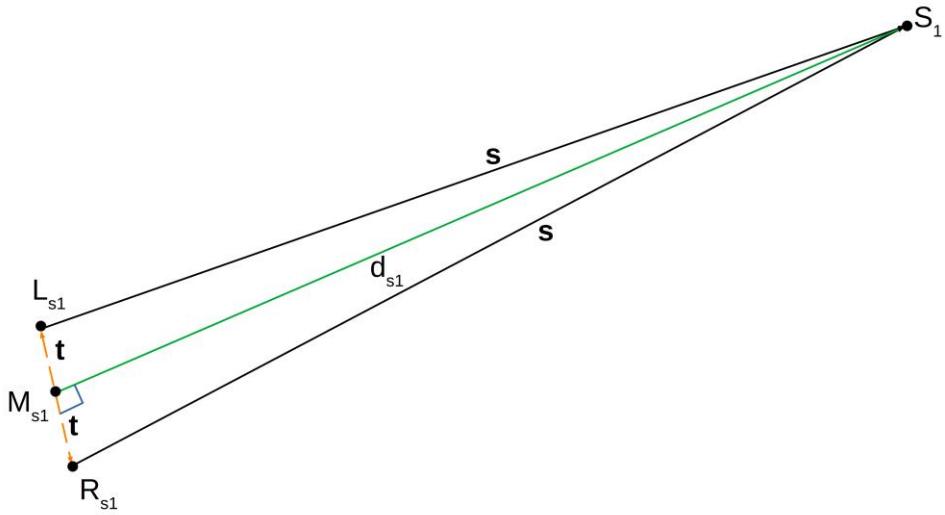
in 3D computer graphics and computer vision. We can see an example of a depth map in the bottom left corner of the above image.

### Depth Calculation

**We assume that we know the distance between cameras and that it's very small compared to the distance between the object and the cameras.** Under that assumption, we can determine the location of the 3D point in space by triangulation. The depth is a perpendicular cast on the line joining the two cameras:



The above image shows the actual depth  $d_{S_1}$  for the point from the line joining the two cameras. Let's note that the angle between the line  $d_{S_1}$  and the line  $L_{S_1}R_{S_1}$  is not exactly 90 degrees. In reality, however, the distance  $L_{S_1}R_{S_1}$  is very small compared to  $d_{S_1}$ . This results in the angle between the line  $d_{S_1}$  and the line  $L_{S_1}R_{S_1}$  being approximately 90 degrees. Since we determined the location of  $S_1$  by triangulation, and we know the relative distance  $L_{S_1}R_{S_1}$ , we can calculate the depth  $d_{S_1}$  using the Pythagorean theorem:



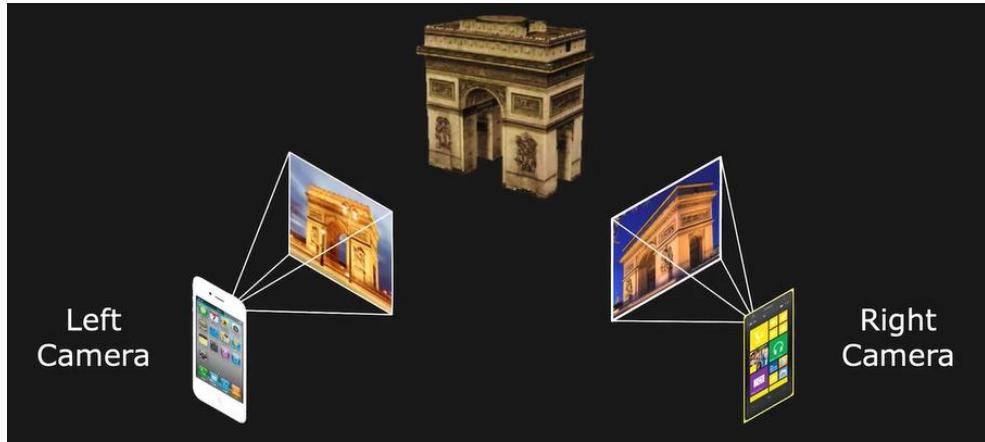
Since  $s$  is very large compared to  $t$ , the angle  $\angle S_1 R_{S_1} M_{S_1}$  approaches  $90^\circ$ . Lengths  $L_{S_1}M_{S_1}$  and  $M_{S_1}R_{S_1}$  are almost the same (denoted by  $t$ ). Also, lengths  $L_{S_1}S_1$  and  $R_{S_1}S_1$  are almost the same (denoted by  $s$ ). Applying the Pythagorean theorem, we get  $s^2 = d_{S_1}^2 + t^2$ . Solving for the depth of point  $S_1$  we get:

$$d_{S_1} = \sqrt{s^2 - t^2}$$

Since  $s$  is very large compared to  $t$ , the depth  $d_{S_1}$  is close to  $s$ .

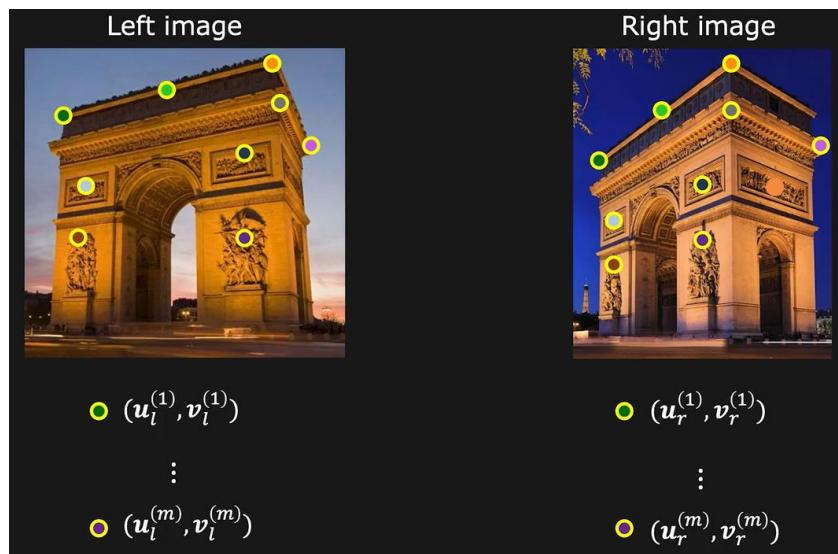
## Uncalibrated Stereo (Triangulation)

The problem of computing placement of points in 3D space, given two uncalibrated perspective views is called **Uncalibrated Stereo** problem. This is a method to estimate 3D structure of a static scene from two arbitrary views.



We assume that the **Intrinsic parameters** ( $f_x, f_y, o_x, o_y$ ) are known for both views/cameras. We want to find the **Extrinsic parameters** (relative position / orientation of cameras).

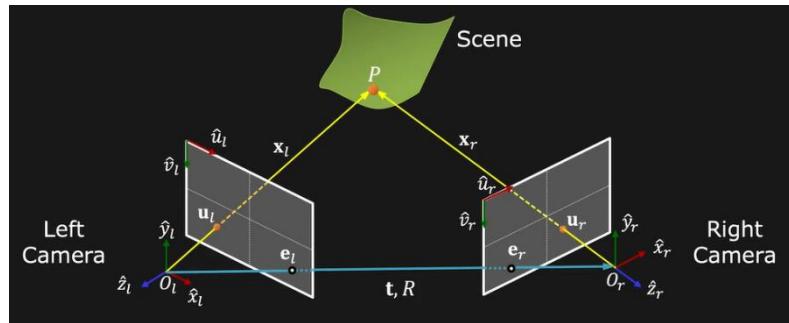
- First, we are assuming that the **Camera Matrix  $K$**  is known for each camera.
- Second, we need to find few **Reliable Corresponding Points**. We find a set of **corresponding features** (at least 8) in left and right images (e.g. using SIFT or hand-picked).



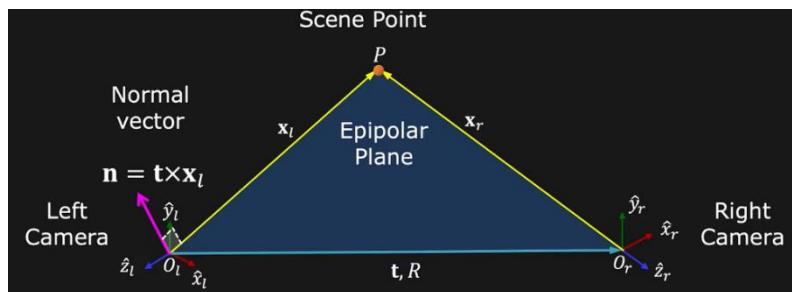
- Third, we find **Relative Camera Position  $t$**  and **Orientation  $R$** .
- Fourth, we find the **Dense Correspondence** (for every point in left image, find the corresponding point in right image).
- Fifth, we compute **Depth** using **Triangulation**.

## EPIPOLAR GEOMETRY

**Epipole** is an image point of origin/pinhole of one camera as viewed by the other camera. In the following example  $e_l$  and  $e_r$  are the epipoles (on the blue line that connects between the projection of the center of the left camera onto the right image and the projection of the center of the right camera onto the left image). Those points are unique for a given stereo pair.

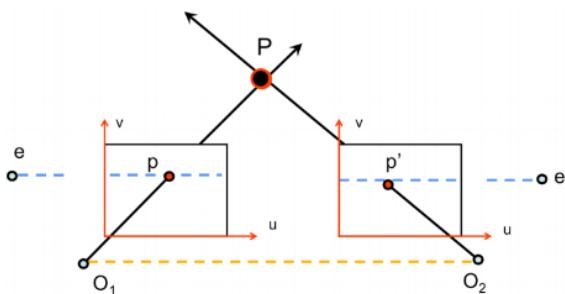


Epipolar Plane or Scene Point  $P$  is the plane formed by camera origins ( $O_l$  and  $O_r$ ), the epipoles ( $e_l$  and  $e_r$ ) and the scene point  $P$ . Every scene point lies on a **unique epipolar plane**.



The lines defined by the intersection of the epipolar plane and the two image planes are known as the **epipolar lines**. The epipolar lines have the property that they intersect the baseline at the respective epipoles in the image plane.

When the two image planes are parallel, then the epipoles  $e_l$  and  $e_r$  are located at infinity. Notice that the epipolar lines are parallel to the  $u$  axis of each image plane.



An interesting case of epipolar geometry is shown above, which occurs when the image planes are parallel to each other. When the image planes are parallel to each other, then the epipoles  $e_l$  and  $e_r$  will be located at infinity since the baseline joining the centers  $O_l, O_r$  is parallel to the image planes. Another important byproduct of this case is that the epipolar lines are parallel to an axis of each image plane. This case is especially useful in **Image Rectification**.

The normal vector to the epipolar plane is:

$$n = t \times x_l$$

Reminder for cross product:

$$s = a \times b = s_1 i + s_2 j + s_3 k$$

$$\begin{aligned}s_1 &= a_2 b_3 - a_3 b_2 \\s_2 &= a_3 b_1 - a_1 b_3 \\s_3 &= a_1 b_2 - a_2 b_1\end{aligned}$$

Where the dot product of  $n$  and  $x_l$  (perpendicular vectors) is zero:

$$x_l \cdot n = x_l \cdot (t \times x_l) = 0$$

That we can write in matrix form (this is the **epipolar constraint** -  $\tilde{x}_l^T E \tilde{x}_r = 0$ ):

$$[x_l \ y_l \ z_l] \begin{bmatrix} t_y z_l - t_z y_l \\ t_z x_l - t_x z_l \\ t_x y_l - t_y x_l \end{bmatrix} = 0 \Rightarrow [x_l \ y_l \ z_l] \underbrace{\begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}}_{T_X} \begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix} = 0$$

Where  $T_X$  is the translation matrix, remembering that we can show the points on the left picture as function of points on right picture using:

- $t_{3 \times 1}$  – Position of Right Camera in Left Camera's Frame
- $R_{3 \times 3}$  – Orientation of Left Camera in Right Camera's Frame

Using the transformation of points:

$$x_l = Rx_r + t \Rightarrow \begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Substituting into the epipolar constraint gives:

$$\begin{aligned}[x_l \ y_l \ z_l] \left( \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} + \underbrace{\begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}}_{t \times t = 0} \right) &= 0 \\[x_l \ y_l \ z_l] \left( \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} \right) &= 0\end{aligned}$$

We will define an **Essential Matrix  $E$**  (depends only on external parameters), such that:

$$E = T_X R \Rightarrow \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

Given that  $T_x$  is a **Skew-Symmetric** matrix ( $a_{ij} = -a_{ji}$ ) and  $R$  is **orthonormal** matrix, it is possible to “**decouple**”  $T_x$  and  $R$  from their product using “**Singular Value Decomposition**”.

So if  $E$  is known, we can calculate  $t$  and  $R$ .

Hence:

$$x_l^T E x_r = 0 \Rightarrow [x_l \ y_l \ z_l] \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} = 0$$

But unfortunately, we don't have  $x_l$  and  $x_r$ , we do have the projections of the point in the scene onto the images with following perspective projection equations (for left camera):

$$\begin{cases} u_l = f_x^{(l)} \frac{x_l}{z_l} + o_x^{(l)} \\ v_l = f_y^{(l)} \frac{y_l}{z_l} + o_y^{(l)} \end{cases} \Rightarrow \begin{cases} z_l u_l = f_x^{(l)} x_l + z_l o_x^{(l)} \\ z_l v_l = f_y^{(l)} y_l + z_l o_y^{(l)} \end{cases}$$

And using homogeneous coordinates we can represent in matrix form:

$$z_l \begin{bmatrix} u_l \\ v_l \\ 1 \end{bmatrix} = \begin{bmatrix} z_l u_l \\ z_l v_l \\ z_l \end{bmatrix} = \begin{bmatrix} f_x^{(l)} x_l + z_l o_x^{(l)} \\ f_y^{(l)} y_l + z_l o_y^{(l)} \\ z_l \end{bmatrix} = \begin{bmatrix} f_x^{(l)} & 0 & o_x^{(l)} \\ 0 & f_y^{(l)} & o_y^{(l)} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix}$$

Where the intrinsic parameters  $(f_x^{(l)}, f_y^{(l)}, o_x^{(l)}, o_y^{(l)})$  are known, and we define the left camera matrix  $K_l$ :

$$K_l = \begin{bmatrix} f_x^{(l)} & 0 & o_x^{(l)} \\ 0 & f_y^{(l)} & o_y^{(l)} \\ 0 & 0 & 1 \end{bmatrix}$$

We can do the same for right camera and get the right camera matrix  $K_r$ :

$$z_r \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = \begin{bmatrix} f_x^{(r)} & 0 & o_x^{(r)} \\ 0 & f_y^{(r)} & o_y^{(r)} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix}$$

Where:

$$K_r = \begin{bmatrix} f_x^{(r)} & 0 & o_x^{(r)} \\ 0 & f_y^{(r)} & o_y^{(r)} \\ 0 & 0 & 1 \end{bmatrix}$$

In a more compact form:

$$x_l^T = [u_l \ v_l \ 1] z_l K_l^{-1} {}^T, \quad x_r = K_r^{-1} z_r \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix}$$

Substituting in the epipolar equation:

$$x_l^T E x_r = 0 \Rightarrow [x_l \ y_l \ z_l] \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} = 0 \Rightarrow$$

$$[u_l \ v_l \ 1] z_l K_l^{-1 T} \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} K_r^{-1} z_r \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = 0$$

$z_l$  and  $z_r$  is the depth and it can't be zero:

$$z_l, z_r \neq 0$$

So, we can eliminate  $z_l$  and  $z_r$  from the equation above and get:

$$[u_l \ v_l \ 1]^T K_l^{-1 T} \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} K_r^{-1} \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = 0$$

Rewriting in terms of image coordinates (that are known to us):

$$[u_l^{(i)} \ v_l^{(i)} \ 1]^T \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = 0$$

Where we define the **Fundamental Matrix  $F$** , such that:

$$F = K_l^{-1 T} E K_r^{-1} = K_l^{-1 T} \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} K_r^{-1} = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}$$

So, if we find the fundamental matrix, we can find the essential matrix:

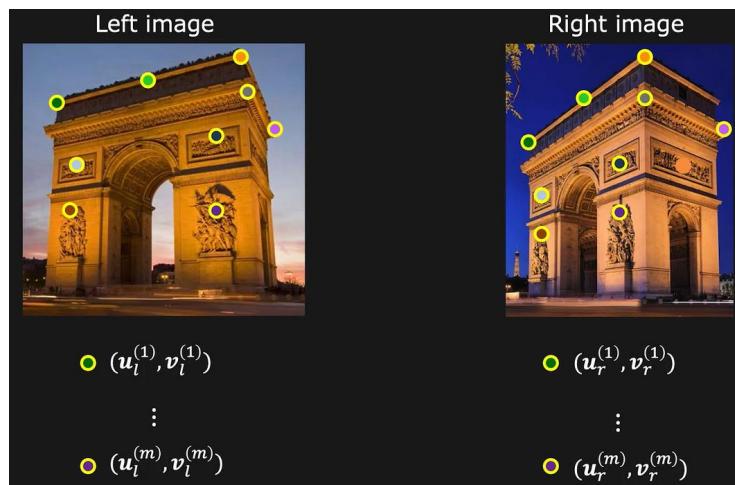
$$E = K_l F K_r$$

Once we have found the essential matrix, we can decouple it using SVD to get:

$$E = T_x R$$

## STEREO CALIBRATION (ESTIMATING THE FUNDAMENTAL MATRIX)

First we find a set of **corresponding features** in left and right images (e.g. using SIFT or hand-picked). The following algorithm is also known as **Eight-Point Algorithm**.



- A. For each corresponding  $i$ , write out epipolar constraint:

$$\begin{bmatrix} u_l^{(i)} & v_l^{(i)} & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} u_r^{(i)} \\ v_r^{(i)} \\ 1 \end{bmatrix} = 0$$

- B. Rearrange term to form a linear system:

$$\underbrace{\begin{bmatrix} u_l^{(1)}u_r^{(1)} & u_l^{(1)}v_r^{(1)} & u_l^{(1)} & v_l^{(1)}u_r^{(1)} & v_l^{(1)}v_r^{(1)} & v_l^{(1)} & u_r^{(1)} & v_r^{(1)} & 1 \\ \vdots & \vdots \\ u_l^{(i)}u_r^{(i)} & u_l^{(i)}v_r^{(i)} & u_l^{(i)} & v_l^{(i)}u_r^{(i)} & v_l^{(i)}v_r^{(i)} & v_l^{(i)} & u_r^{(i)} & v_r^{(i)} & 1 \\ \vdots & \vdots \\ u_l^{(n)}u_r^{(n)} & u_l^{(n)}v_r^{(n)} & u_l^{(n)} & v_l^{(n)}u_r^{(n)} & v_l^{(n)}v_r^{(n)} & v_l^{(n)} & u_r^{(n)} & v_r^{(n)} & 1 \end{bmatrix}}_{A \text{ Known}} = \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix}_{f \text{ Unknown}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

We get:

$$Af = 0$$

Remembering that fundamental matrix  $F$  and  $kF$  describe the same epipolar geometry. That is,  $F$  is defined only up to a scale. So we set the fundamental matrix to some arbitrary scale, like:

$$\|f\|^2 = 1$$

- C. Solving for  $F$ , we find **least squares** solution for fundamental matrix  $F$ . We want  $Af$  as close to 0 as possible and  $\|f\|^2 = 1$ , getting the following optimization problem:

$$\min_f \|Af\|^2 \text{ such that } \|f\|^2 = 1$$

This is a constrained linear least squares problem.

Applying SVD to  $A$  yields the decomposition  $U\Sigma V^T$  with  $U$  and  $V$  orthonormal matrices and  $\Sigma$  a diagonal matrix containing the singular values. These singular values  $\sigma_i$  are positive and in decreasing order. Therefore in our case  $\sigma_9$  is guaranteed to be identically zero (8 equations for 9 unknowns) and thus the last column of  $V$  is the correct solution (at least as long as the eight equations are linearly independent, which is equivalent to all other singular values being non-zero).

We then rearrange the vector  $f$  to get the fundamental matrix  $F$ . However, in the presence of noise, this matrix will not satisfy the rank-2 constraint. This means that there will not be real epipoles through which all epipolar lines pass, but that these will be "smeared out" to a small region. A solution to this problem is to obtain  $F$  as the closest rank-2 approximation of the solution coming out of the linear equations.

- D. We compute essential matrix  $E$  from known left and right intrinsic camera matrices and fundamental matrix  $F$ :

$$E = K_l^T F K_r$$

- E. Extract  $R$  and  $t$  from  $E$  (using SVD):

$$E = T_x R$$

*Seven-Point Algorithm*

In fact, the fundamental matrix only has seven degrees of freedom. If one is prepared to solve non-linear equations, seven points must thus be sufficient to solve for it. In this case the rank-2 constraint must be enforced during the computations.

A similar approach as in the previous section can be followed to characterize the right null-space of the system of linear equations originating from the seven point correspondences. This space can be parameterized as follows:

$$\nu_1 + \lambda\nu_2 \text{ (or } F_1 + \lambda F_2\text{)}$$

With  $\nu_1$  and  $\nu_2$  being the two last columns of  $V$  (obtained through SVD) and  $F_1$  respectively  $F_2$  the corresponding matrices. The rank-2 constraint is then written as:

$$\det(F_1 + \lambda F_2) = a_3\lambda^3 + a_2\lambda^2 + a_1\lambda + a_0 = 0$$

Which is a polynomial of degree 3 in  $\lambda$ . This can simply be solved analytically. There are always 1 or 3 real solutions. The special case  $F_1$  (which is not covered by this parameterization) is easily checked separately, i.e. it should have rank-2. If more than one solution is obtained then more points are needed to obtain the true fundamental matrix.

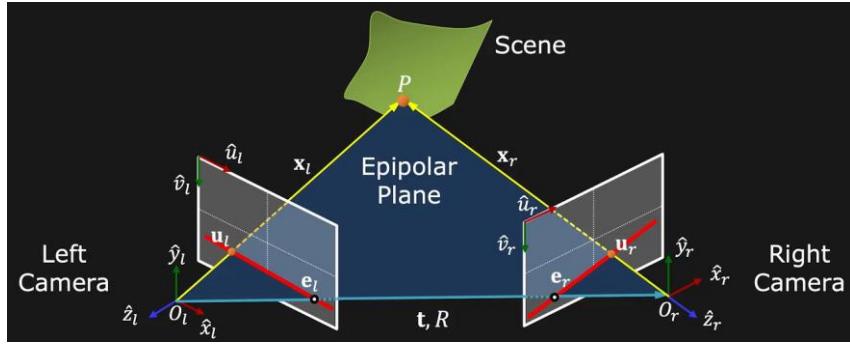
*More Points*

It is clear that when more point matches are available the redundancy should be used to minimize the effect of the noise. The eight-point algorithm can easily be extended to be used with more points. In this case the matrix  $A$  of  $Af = 0$  will be much bigger, it will have one row per point match. The solution can be obtained in the same way, but in this case the last singular value will not be perfectly equal to zero. It has been pointed out that in practice it is very important to normalize the equations. This is for example achieved by transforming the image to the interval  $[-1,1] \times [-1,1]$  so that all elements of the matrix  $A$  are of the same order of magnitude.

**FINDING CORRESPONDENCES**

In the case where the right camera is just like the left camera only translated along the horizontal direction (by some distance  $b$ ), the corresponding scene points lie on the **same horizontal scan-line**. So, finding correspondence is a **1D search**. In the case of uncalibrated stereo, once we found the rotation and translation between two cameras, the stereo matching problem remain a 1D search problem (but along which line?).

Every scene point has two **corresponding epipolar lines**, one each on the two image planes. Given a point in one image, the corresponding point in the other image must lie on the epipolar line. So finding correspondence reduces to a 1D search, along the epipolar line (the red one).



Using the epipolar constraint equation:

$$\underbrace{\begin{bmatrix} u_l & v_l & 1 \end{bmatrix}}_{Known} \underbrace{\begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}}_{Known} \underbrace{\begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix}}_{Unknown} = 0$$

Expanding the matrix equation gives:

$$(f_{11}u_l + f_{21}v_l + f_{31})u_r + (f_{12}u_l + f_{22}v_l + f_{32})v_r + (f_{13}u_l + f_{23}v_l + f_{33}) = 0$$

We get the equation for **right epipolar line**:

$$a_l u_r + b_l v_r + c_l = 0$$

Similarly we can calculate epipolar line in the left image for a point in right image.

### *Essential Matrix vs. Homography*

The **essential matrix maps a point to a line** (to an epipolar line):

$$\tilde{l}_r = E\tilde{x}_l$$

where is the **homography maps a point to point**.

Homography applies only for planar scenes.

The essential matrix has 5 DoFs: 3 DoF for rotation and 3 DoF for translation, up to a scale so 1 DoF is removed.

### *Essential Matrix Properties*

- $x_r^T E x_l = 0$
- $x_l^T l_l = 0, x_r^T l_r = 0$
- $l_r = E x_l, l_l = E^T x_r$
- $e_r^T E = 0, E e_l = 0$

### *Fundamental Matrix vs. Essential matrix*

**Fundamental matrix** encodes the relationship of two views in the image coordinate with **7 degree of freedom**. **Essential matrix** is a special case of fundamental matrix which only encodes the relative motion of two cameras in the camera coordinates, depending only on the relative rotation  $R$  and translation  $t$ , which results in **degree of freedom of 5**.

*Different Matrices (Fundamental, Essential, Homography and Projection)*

Fundamental Matrix, Essential Matrix, Homography, and Projection Matrix are all used in computer vision to relate 2D image coordinates to 3D world coordinates. However, they differ in their purpose and the type of relationship they represent.

- **Fundamental Matrix:** The fundamental matrix is used to establish the correspondence between two views of a scene taken by two different cameras (deals with uncalibrated cameras). It represents the relationship between points in two different views, allowing the computation of the epipolar lines (lines in one image along which the corresponding point in the other image must lie). The fundamental matrix has 7 degrees of freedom (two for each of the epipoles and three for the homography between the two pencils of epipolar lines), its rank is always two and is typically estimated from point correspondences between the two images.
- **Essential Matrix:** The essential matrix is a related concept to the fundamental matrix. It is a 3x3 matrix that encodes the geometric relationship between two camera views in a **normalized coordinate system** (deals with calibrated cameras). The essential matrix is derived from the fundamental matrix, but it is invariant to changes in the camera's intrinsic parameters. The essential matrix has 5 degrees of freedom (three for rotation and two for the direction of translation -- the magnitude of translation cannot be recovered due to the depth/speed ambiguity) and is typically estimated from point correspondences between the two images.
- **Homography:** A homography is a transformation that maps points in one image to points in another image of the same plane. It represents the relationship between two views of a planar object or a scene in which the camera's translation is negligible. A homography is a 3x3 matrix and has 8 degrees of freedom. It can be estimated from point correspondences between the two images. There are only two cases where the transformation between two views is a projective transformation (i.e. a homography): either **the scene is planar** or the **two views were generated by a camera rotating around its center**.
- **Projection Matrix:** The projection matrix is used to relate 3D points in the world coordinate system to their corresponding 2D image points in the camera's image plane. It represents the transformation from the world coordinate system to the camera coordinate system. The projection matrix is a 3x4 matrix that consists of the camera's intrinsic parameters (focal length, principal point) and extrinsic parameters (rotation and translation). It has 11 degrees of freedom and can be estimated using camera calibration techniques.

In summary, the fundamental matrix and essential matrix are used to relate two views of a scene taken by two different cameras, while the homography is used to relate two views of a planar object or scene. The projection matrix is used to relate 3D points in the world coordinate system to their corresponding 2D image points in the camera's image plane.

## COMPUTING DEPTH

Given the intrinsic parameters, the projections of scene point on the two image sensors are the left and right imaging equations:

$$\begin{bmatrix} u_l \\ v_l \\ 1 \end{bmatrix} = \begin{bmatrix} f_x^{(l)} & 0 & o_x^{(l)} & 0 \\ 0 & f_y^{(l)} & o_y^{(l)} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_l \\ y_l \\ z_l \\ 1 \end{bmatrix}, \quad \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = \begin{bmatrix} f_x^{(r)} & 0 & o_x^{(r)} & 0 \\ 0 & f_y^{(r)} & o_y^{(r)} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix}$$

We also know the relative position and orientation between the two cameras (from calibration):

$$\begin{bmatrix} x_l \\ y_l \\ z_l \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix}$$

Substituting only to the left imaging equation we get:

$$\begin{bmatrix} u_l \\ v_l \\ 1 \end{bmatrix} = \begin{bmatrix} f_x^{(l)} & 0 & o_x^{(l)} & 0 \\ 0 & f_y^{(l)} & o_y^{(l)} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix}, \quad \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = \begin{bmatrix} f_x^{(r)} & 0 & o_x^{(r)} & 0 \\ 0 & f_y^{(r)} & o_y^{(r)} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix}$$

Rearranged to matrix form:

$$\tilde{u}_l = P_l \tilde{x}_r, \quad \tilde{u}_r = M_{int_r} \tilde{x}_r$$

The imaging equations:

$$\begin{aligned} \tilde{u}_r = M_{int_r} \tilde{x}_r \Rightarrow \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} &= \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix} \\ \tilde{u}_l = P_l \tilde{x}_r \Rightarrow \begin{bmatrix} u_l \\ v_l \\ 1 \end{bmatrix} &= \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix} \end{aligned}$$

Rearranging the terms:

$$\underbrace{\begin{bmatrix} u_r m_{31} - m_{11} & u_r m_{32} - m_{12} & u_r m_{33} - m_{13} \\ v_r m_{31} - m_{21} & v_r m_{32} - m_{22} & v_r m_{33} - m_{23} \\ u_l p_{31} - p_{11} & u_l p_{32} - p_{12} & u_l p_{33} - p_{13} \\ v_l p_{31} - p_{21} & v_l p_{32} - p_{22} & v_l p_{33} - p_{23} \end{bmatrix}}_{\substack{A_{4 \times 3} \\ Known}} \underbrace{\begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix}}_{\substack{Unknown}} = \underbrace{\begin{bmatrix} m_{14} - m_{34} \\ m_{24} - m_{34} \\ p_{14} - p_{34} \\ p_{24} - p_{34} \end{bmatrix}}_{\substack{b_{4 \times 1} \\ Known}}$$

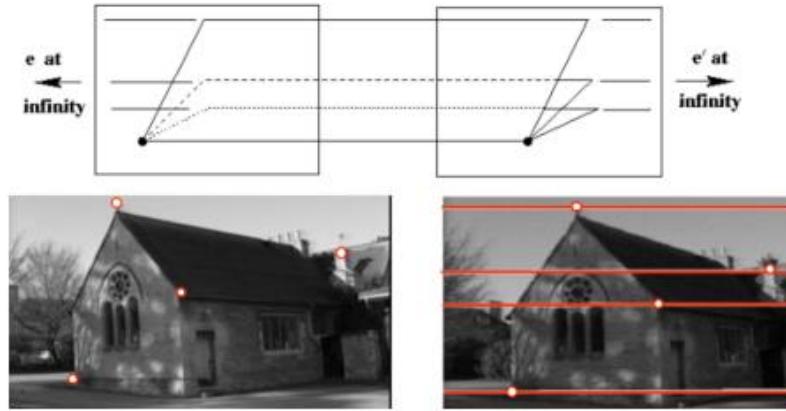
We will find **least squares solution** using **pseudo-inverse**:

$$Ax_r = b \Rightarrow A^T Ax_r = A^T b \Rightarrow x_r = (A^T A)^{-1} A^T b$$

## IMAGE RECTIFICATION

Recall that an interesting case for epipolar geometry occurs when two images are parallel to each other.

If both cameras face exactly the same direction, their image planes will be co-planar and parallel to the baseline. This means that the epipoles lie at infinity (outside the image) and all epipolar lines are parallel to each other and horizontal on the images. In this setting, correspondence search can be done along horizontal scanlines which simplifies the implementation.



If both images aren't in the required setup, We can re warp the images through rotation, mapping both image planes to a common plane parallel to the baseline. This process is called rectification and does not require knowledge about the 3D structure. Similar to the homography for stitching panoramas, the rotation around the camera center can be computed by estimating a rotation homography.

Let us first compute the Essential matrix  $E$  in the case of **parallel image planes**. We can assume that the two cameras have the same  $K$  and that there is no relative rotation between the cameras ( $R = I$ ). In this case, let us assume that there is only a translation along the  $x$  axis, giving  $t = (t_x, 0, 0)$ . This gives:

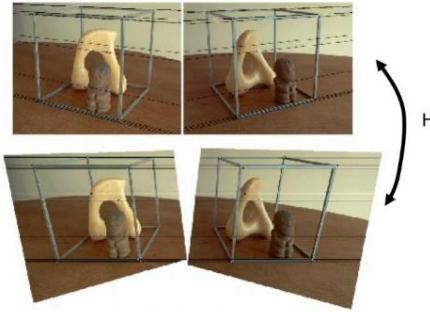
$$E = T_x R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix}$$

Once  $E$  is known, we can find the directions of the epipolar lines associated with points in the image planes. Let us compute the direction of the epipolar line  $l_l$  associated with point  $x_r$ :

$$l_r = Ex_l = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} \begin{bmatrix} u_l \\ v_l \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -t_x \\ t_x v_l \end{bmatrix}$$

We can see that the direction of  $l_l$  is horizontal, as is the direction of  $l_r$ , which is computed in a similar manner.

The process of image rectification involves computing two homographies that we can apply to a pair of images to make them parallel.

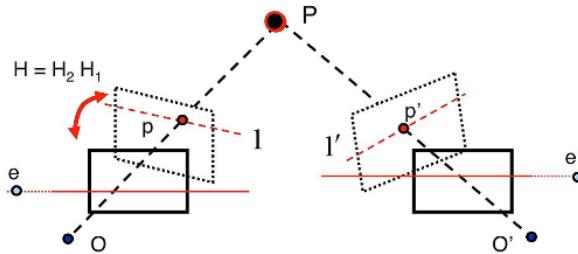


If we use the **epipolar constraint**:

$$x_r^T E x_l = 0 \Rightarrow x_r^T \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} x_l = x_r^T \begin{pmatrix} 0 \\ -t_x \\ t_x v_l \end{pmatrix} = (u_r \ v_r \ 1) \begin{pmatrix} 0 \\ -t_x \\ t_x v_l \end{pmatrix} = t_x v_l - t_x v_r = t_x(v_l - v_r) = 0 \rightarrow v_l = v_r$$

Then we arrive at the fact that  $v_l = v_r$ , demonstrating that  $x_l$  and  $x_r$  share the same  $v$ -coordinate. Consequently, there exists a very straightforward relationship between the corresponding points. Therefore, rectification, or the process of making any two given images parallel, becomes **useful when discerning the relationships between corresponding points in images**.

The **rectification problem setup**: we compute two homographies that we can apply to the image planes to make the resulting planes parallel:



In order to rectify an image, we calculate a rectifying rotation matrix  $R_{rect} = (r_1, r_2, r_3)^T$  that rotates the cameras virtually such that the above setup is fulfilled, i.e. aligning the x-axis of the camera coordinate system with the translation vector while introducing as little distortion to the other two axes as possible (these must be perpendicular to each other). One simple choice for  $R_{rect}$  is the following:

First, we make sure the new x-axis aligns with the translation vector by setting  $r_1$  to its normalized form:

$$r_1 = \frac{t}{\|t\|}$$

Next, we choose  $r_2$  such that the new y-axis is perpendicular to the new x-axis and perpendicular to the old z-axis (this helps to keep the distortion small):

$$r_2 = [(0,0,1)^T] \times r_1$$

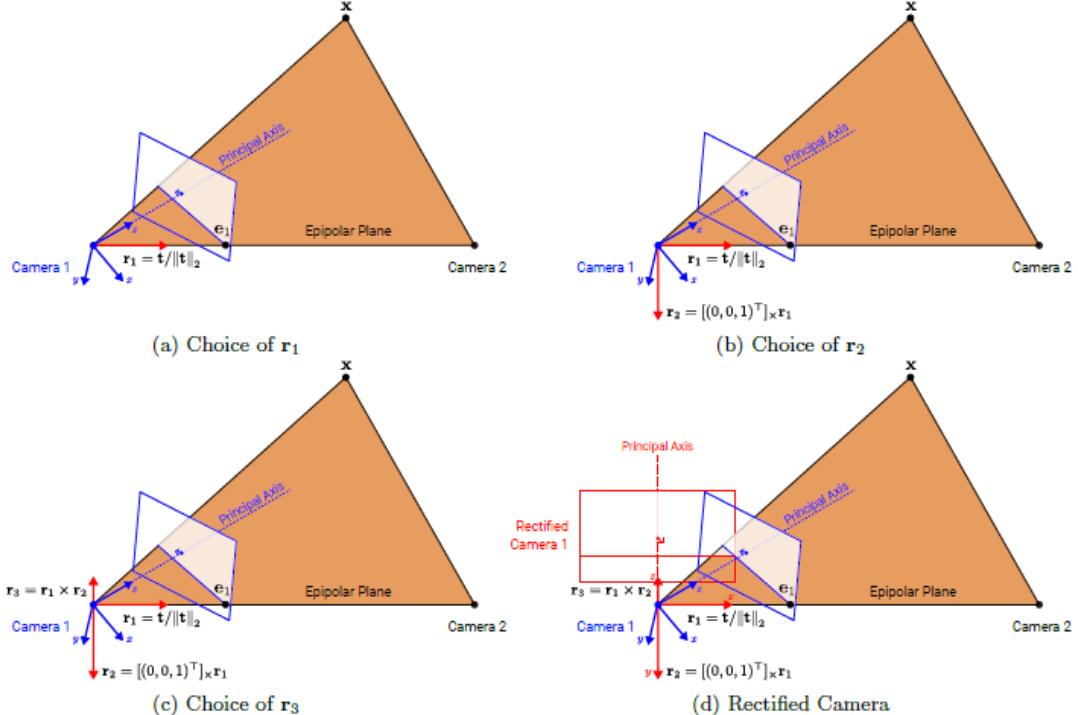
Finally,  $r_3$  should be perpendicular to  $r_1$  and  $r_2$ :

$$r_3 = r_1 \times r_2$$

As the epipole in the first image is in the direction of  $r_1$ , it is easy to see that the rotated epipole is ideal, i.e. it lies at infinity, in the x-direction:

$$R_{rect}r_1 = (1,0,0)^T$$

Thus, applying  $R_{rect}$  to the first camera leads to parallel and horizontal epipolar lines.



### Rectification algorithm:

1. Estimate essential matrix  $E$ , decompose into  $t$  and  $R$ , and construct  $R_{rect}$  as above.
2. Warp pixels in the first image as follows:

$$x'_l = K R_{rect} K^{-1} x_l$$

3. Warp pixels in the second image as follows:

$$x'_r = K R R_{rect} K^{-1} x_r$$

Rectifying a pair of images does not require knowledge of the two camera matrices  $K_l$ ,  $K_r$  or the relative transformation  $R$ ,  $t$  between them. Instead, we can use the **Fundamental matrix** estimated by the **Normalized Eight Point Algorithm**. Upon getting the Fundamental matrix, we can compute the epipolar lines  $l_l^{(i)}$  and  $l_r^{(i)}$  for each correspondence  $x_l^{(i)}$  and  $x_r^{(i)}$ .

From the set of epipolar lines, we can then estimate the epipoles  $e_l$  and  $e_r$  of each image. This is because we know that the epipole lies in the intersection of all the epipolar lines. In the real

world, due to noisy measurements, all the epipolar lines will not intersect in a single point. Therefore, computing the epipole can be found by **minimizing the least squared error** of fitting a point to all the epipolar lines.

### Examples – Epipolar Geometry Questions

#### 1. Parallel camera planes

Suppose we know that the camera planes are parallel (after rectification - i.e. they are only translated on the x axis). Does that information allow us to use less sets of matching points to find?

#### Solution:

The camera matrices are given by  $M_l = K_l[I|0]$  and  $M_r = K_r[R|0]$ , where:

$$K_l = \begin{bmatrix} f_x^l & 0 & 0 \\ 0 & f_y^l & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad K_r = \begin{bmatrix} f_x^r & 0 & 0 \\ 0 & f_y^r & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad R = I, \quad t = \begin{bmatrix} t_x \\ 0 \\ 0 \end{bmatrix}$$

We know that  $F = (K_l^T)^{-1} E K_r^{-1} = (K_l^T)^{-1} (T_x R) K_r^{-1} = (K_l^T)^{-1} (T_x I) K_r^{-1}$ , so:

$$K_l^T = \begin{bmatrix} f_x^l & 0 & 0 \\ 0 & f_y^l & 0 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow (K_l^T)^{-1} = \begin{bmatrix} \frac{1}{f_x^l} & 0 & 0 \\ 0 & \frac{1}{f_y^l} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$K_r = \begin{bmatrix} f_x^r & 0 & 0 \\ 0 & f_y^r & 0 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow (K_r)^{-1} = \begin{bmatrix} \frac{1}{f_x^r} & 0 & 0 \\ 0 & \frac{1}{f_y^r} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Remember that  $T_x = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}$  and for our case  $T_x = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix}$ , hence:

$$E = T_x R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix}$$

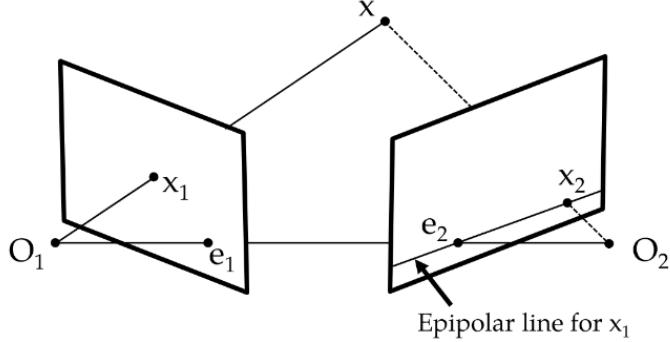
And:

$$F = \begin{bmatrix} \frac{1}{f_x^l} & 0 & 0 \\ 0 & \frac{1}{f_y^l} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{f_x^r} & 0 & 0 \\ 0 & \frac{1}{f_y^r} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{f_x^l} & 0 & 0 \\ 0 & \frac{1}{f_y^l} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & \frac{t_x}{f_y^r} & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -\frac{t_x}{f_y^l} \\ 0 & \frac{t_x}{f_y^r} & 0 \end{bmatrix}$$

$F$  now has only 2 DoFs, so we only need 2 points.

**Note:** Even if we look at more complex intrinsic matrices, and 3 degrees of translation, we can still spare 3 DoFs (and 3 points) by not having a rotation matrix!

## 2. Epipoles



Let  $M_1$  and  $M_2$  be two camera matrices. We know that the fundamental matrix corresponding to these camera matrices is of the following form:

$$F = [a]_{\times}A$$

Where  $[a]_{\times}$  is the matrix:

$$[a]_{\times} = \begin{pmatrix} 0 & a_y & -a_z \\ -a_y & 0 & a_x \\ a_z & -a_x & 0 \end{pmatrix}$$

Assume that  $M_1 = [I|0]$  and  $M_2 = [A|a]$ , where  $A$  is a  $3 \times 3$  (nonsingular) matrix. Prove that a column of  $M_2$  denoted by  $a$ .

**Solution:**

We know that two epipoles of any stereo system can be expressed as:

$$\begin{aligned} Fe_r &= 0 \\ F &= [a]_{\times}A \end{aligned}$$

And:

$$\begin{aligned} Fe_l &= 0 \\ F^T &= A^T[a]_{\times}^T \end{aligned}$$

Because  $[a]_{\times}$  is skew-symmetric,  $[a]_{\times}^T = -[a]_{\times}$ . Thus, we can simply plug in  $a$  in for  $e_l$  and  $e_r$ :

$$F^T a = A^T[a]_{\times}^T a = -A^T[a]_{\times} a = -A^T 0$$

Because  $[a]_{\times} a = 0$ ,  $a$  must clearly be the right epipole.

## 3. Epipolar geometry

Consider the point  $X$  expressed with respect to the world coordinate system. Suppose that the transformation between two cameras is described by a rotation matrix  $R$  and a translation vector  $t$  so that  $X' = RX + t$ . Assume that the first camera coordinate system is equivalent to the world coordinate system and that its camera matrix is  $K$ . The camera calibration matrix of the second camera is  $K'$ .

- a. Write the camera projection matrices  $M$  and  $M'$  in terms of  $K$ ,  $K'$ ,  $R$ , and  $t$ .
- b. What are the expressions for the epipoles,  $e$  and  $e'$  in terms of one or more of the following:  $M$ ,  $M'$ ,  $R$  and  $t$ .
- c. An alternate formulation of the fundamental matrix,  $F = [e']_x K' R K^{-1}$ . Using the results from the earlier parts, show that  $e' \times x' = Fx$  where  $x'$  is the image of  $X$  in the second camera coordinate system.
- d. What is the geometric interpretation of  $e' \times x'$ ?

**Solution:**

- a. The camera projection matrices:

$$\begin{aligned} M &= K[I|0] \\ M' &= K'[R|t] \end{aligned}$$

- b. We first find the camera centers:

$$O = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

And the center of the second camera, translated from its coordinate system to world coordinate system:

$$O' = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \Rightarrow O' = RX + t \Rightarrow O' - t = RX \Rightarrow -t = RX \Rightarrow -R^{-1}t = X \Rightarrow$$

Remember that **the inverse of rotation matrix is its transpose**:

$$R^T = R^{-1}$$

Hence:

$$X = -R^T t$$

And the second center is:

$$O' = -R^T t$$

The epipoles are the projections of the other camera center:

$$\begin{aligned} e &= M \begin{pmatrix} O' \\ 1 \end{pmatrix} = M \begin{pmatrix} -R^T t \\ 1 \end{pmatrix} = -KR^T t \\ e' &= M' \begin{pmatrix} O \\ 1 \end{pmatrix} = K't \end{aligned}$$

- c. Using the new formulation:

$$Fx = [e']_x K' R K^{-1} x \Rightarrow$$

Where  $x = MX$  and hence:

$$\Rightarrow [e']_x K' R K^{-1} M X = [e']_x K' R K^{-1} K[I|0] X = [e']_x K' R[I|0] X = [e']_x K' [R|0] X$$

Now:

$$e' \times x' = e' \times (M' X) = e' \times (K'[R|t] X) \Rightarrow$$

We can use the useful trick that:

$$[R|t] X = R[I|0] X + t$$

Hence:

$$\Rightarrow e' \times (K'(R[I|0] X + t)) = e' \times (K' R[I|0] X + K' t) \Rightarrow$$

We recall that in Part b we showed that  $e' = K' t$  and that the cross product of a vector with itself is zero. Therefore:

$$\Rightarrow e' \times (K'R[I|0]X + e') = e' \times K'R[I|0]X \Rightarrow$$

Where  $[e']_x$  is:

$$[e']_x = \begin{bmatrix} 0 & -e'_z & e'_y \\ e'_z & 0 & -e'_x \\ -e'_y & e'_x & 0 \end{bmatrix}$$

Hence, we get:

$$[e']_x K'[R|0]X$$

And finally:

$$Fx = e' \times x'$$

- d.  $l' = e' \times x'$  is the epipolar line that passes through epipole  $e'$  and point  $x'$  in the image of the second camera.

#### 4. Another Epipolar Geometry

Ada and Charles, young, budding, and aptly-named computer science students, are looking to get involved in the field of computer vision. Naturally, they start by looking at the exciting problem of finding epipoles in images.

**NOTE:** In this problem we will denote Ada with normal variables, and Charles with prime variables (e.g. Ada's camera matrix is  $K$ , Charles' is  $K'$ ).

- a. They decide to use their cellphones to take images. Ada has the hot new "Goosung Y5" and Charles has the slightly older "Huapple C4". They download the phones' camera datasheets and see that:
- Ada's camera has no distortion, no skew, focal length  $\alpha = \beta = 1$ , and principal offset point  $c_x = c_y = 0$ .
  - Charles' phone has all the same parameters, but a skew angle of  $45^\circ$ .

Before doing anything else, Ada and Charles figure that they should write out their camera intrinsic matrices, as it will be useful for later calculations. What are  $K$  and  $K'$ ?

**Hint:**  $\cot(45^\circ) = 1$  and  $\sin(45^\circ) = \frac{1}{\sqrt{2}}$ .

- b. Now that they have their camera matrices written down, they go to take photos of 3D objects. Ada stays in one place and chooses her camera coordinate system to be the same as the world's. She tells Charles to walk a few meters away and up a hill for a better viewpoint. Since Charles kept track of his motion, he knows that his  $R$  and  $T$  relative to Ada are:

$$R = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad T = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Knowing this information, and your answer for their camera matrices, find their Fundamental Matrix  $F$ .

**Hint:** With block algebra, calculating inverses can be done like so:

$$\begin{pmatrix} A & 0 \\ 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & 1 \end{pmatrix}$$

- c. Finally, after taking photos of the 3D object and labeling matching points, they have the following correspondences:

$$p_1 = \begin{pmatrix} 1 \\ \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}, \quad p'_1 = \begin{pmatrix} 3\sqrt{2} \\ 4\sqrt{2} \end{pmatrix}$$

$$p_2 = \begin{pmatrix} 5 \\ \frac{5}{\sqrt{2}} \\ \frac{5}{\sqrt{2}} \end{pmatrix}, \quad p'_2 = \begin{pmatrix} 10\sqrt{2} \\ 0 \end{pmatrix}$$

To reiterate, image points  $p_1$  and  $p_2$  in Ada's image are matched with  $p'_1$  and  $p'_2$  respectively, in Charles' image. Find the location of the epipole in Ada's image plane (ie. its  $x$  and  $y$  coordinates).

**Note:** As much as you'd like to check your  $F$  in (b) with these points, don't. They were chosen irrespective of the answer to the previous section.

**Hint:** Don't get bogged down by complicated expressions. If you find yourself doing lot of arithmetic/algebra, just assign dummy variables to complex expressions and carry on from there.

Getting a correct numerical final value should be a minor concern, showing that you understand the process required to get there is much more important.

- d. Now ignore the previous parts. Assume Ada's camera takes pictures of size  $100 \times 150$  pixels (width by height). If Charles only walked directly in front of Ada along the  $+z$  axis of Ada's image plane and was careful not to rotate relative to Ada's axes, where would the epipole in Ada's image be? Where would the epipole in Charles' image be?

### Solutions:

- a. Intrinsic matrices:

$$K = \begin{pmatrix} \alpha & 0 & c_x \\ 0 & \beta & c_y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = I$$

$$K' = \begin{pmatrix} \alpha & -\alpha \cot(\theta) & c_x \\ 0 & \frac{\beta}{\sin(\theta)} & c_y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 0 \\ 0 & \sqrt{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- b. The fundamental matrix is given by:

$$F = (K^T)^{-1} T_x R K'^{-1} \Rightarrow$$

Compute the inverse of intrinsic matrices:

$$K^T = I \Rightarrow K^{T^{-1}} = I^{-1} = I$$

To compute the inverse of  $K'$  we use block geometry:

$$\begin{pmatrix} 1 & -1 \\ 0 & \sqrt{2} \end{pmatrix}^{-1} = \frac{1}{\sqrt{2}} \begin{pmatrix} \sqrt{2} & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & \frac{1}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} \end{pmatrix}$$

Thus:

$$K'^{-1} = \begin{pmatrix} 1 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Hence, we compute the fundamental matrix:

$$\begin{aligned} \Rightarrow (K^T)^{-1} T_x R K'^{-1} &= \\ &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & -3 & 2 \\ 3 & 0 & -1 \\ -2 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix} = \\ &= \begin{pmatrix} 0 & -3 & 2 \\ 3 & 0 & -1 \\ -2 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -3 & 0 & 2 \\ 0 & -3 & -1 \\ 1 & 2 & 0 \end{pmatrix} \begin{pmatrix} 1 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix} = \\ &= \begin{pmatrix} -3 & -\frac{3}{\sqrt{2}} & 2 \\ 0 & -\frac{3}{\sqrt{2}} & -1 \\ 1 & \frac{3}{\sqrt{2}} & 0 \end{pmatrix} \end{aligned}$$

c. First, using a nice little shortcut, ignoring all the information we have in this question:

$$e = M \begin{bmatrix} -R^T T \\ 1 \end{bmatrix} = K[I|0] \begin{bmatrix} -R^T T \\ 1 \end{bmatrix} = I[I|0] \begin{bmatrix} -R^T T \\ 1 \end{bmatrix} = -R^T T = \begin{bmatrix} -2 \\ 1 \\ -3 \end{bmatrix} = \begin{bmatrix} 2 \\ \frac{3}{3} \\ -\frac{1}{3} \end{bmatrix}$$

The second answer is the more direct intersection of the two epipolar lines. This could have also been computed by going to homogeneous coordinates, computing the cross product of  $l_1$  and  $l_2$ , then dividing by the third element to bring it back to the image coordinate space. In the solutions we do it this way since it's the most basic way of doing it (assumes no knowledge of the cross product).

We know from the course slides that  $l_1 = Fp'_1$  and that  $l_2 = Fp'_2$ . The intersection of these lines is the location of the epipole in Ada's image plane. The other points  $p_1$  and  $p_2$  aren't used at all (since we only asked for Ada's epipole).

Remember, we must work in homogenous coordinates! As such, we append a 1 to the bottom of the given image points.

$$p'_1 = \begin{pmatrix} 3\sqrt{2} \\ 4\sqrt{2} \\ 1 \end{pmatrix}, \quad p'_2 = \begin{pmatrix} 10\sqrt{2} \\ 0 \\ 1 \end{pmatrix}$$

Now we compute  $l_1$  and  $l_2$ :

$$l_1 = \begin{pmatrix} -3 & -\frac{3}{\sqrt{2}} & 2 \\ 0 & -\frac{3}{\sqrt{2}} & -1 \\ 1 & \frac{3}{\sqrt{2}} & 0 \end{pmatrix} \begin{pmatrix} 3\sqrt{2} \\ 4\sqrt{2} \\ 1 \end{pmatrix} = \begin{pmatrix} -9\sqrt{2} - 10 \\ -13 \\ 3\sqrt{2} + 12 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

$$l_2 = \begin{pmatrix} -3 & -\frac{3}{\sqrt{2}} & 2 \\ 0 & -\frac{3}{\sqrt{2}} & -1 \\ 1 & \frac{3}{\sqrt{2}} & 0 \end{pmatrix} \begin{pmatrix} 10\sqrt{2} \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -30\sqrt{2} + 2 \\ -1 \\ 10\sqrt{2} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Now, we rewrite this in a more easy-to-work-with fashion:

$$l_1 \Rightarrow a_1x + a_2y + a_3 = 0$$

$$l_2 \Rightarrow b_1x + b_2y + b_3 = 0$$

Since we're looking for an intercept (that's where the epipole is), we rearrange the equations to get:

$$l_1 \Rightarrow y = \frac{1}{a_2}(-a_1x - a_3)$$

$$l_2 \Rightarrow y = \frac{1}{b_2}(-b_1x - b_3)$$

and equate the  $y$ 's so that we can solve for  $x$  (and plug the result back in to find  $y$ ):

$$\frac{1}{a_2}(-a_1x - a_3) = \frac{1}{b_2}(-b_1x - b_3)$$

$$\frac{a_1}{a_2}x + \frac{a_3}{a_2} = \frac{b_1}{b_2}x + \frac{b_3}{b_2}$$

$$\left(\frac{a_1}{a_2} - \frac{b_1}{b_2}\right)x = \frac{b_3}{b_2} - \frac{a_3}{a_2}$$

$$x = \frac{\frac{b_3}{b_2} - \frac{a_3}{a_2}}{\frac{a_1}{a_2} - \frac{b_1}{b_2}}$$

Finally, we substitute this back into one of the  $l$  equations above to find the  $y$  coordinate (we chose to do  $l_1$ ):

$$y = \frac{1}{a_2} \left( -a_1 \left( \frac{\frac{b_3}{b_2} - \frac{a_3}{a_2}}{\frac{a_1}{a_2} - \frac{b_1}{b_2}} \right) - a_3 \right)$$

**Note:** The answer from this method and the answer from the first method above are different. This is completely fine since these point correspondences were chosen arbitrarily and not with respect to actual image values (if they were, then this answer would match the relatively nice values above).

- d. Since Charles simply walked forward (i.e. performed only a  $z$  translation and no rotation), Ada's epipole will be at the center of her image, at  $x = 50, y = 75$  pixels. Charles' epipole will be at the same position in his image, at  $x = 50, y = 75$  pixels.

## 5. Yet another epipolar geometry

- a. Assume you have two cameras, both with intrinsic parameters and rotations  $K = R = I$ .

Then for each of the three translation vectors  $t_1, t_2, t_3$  given below, compute the essential matrix, describe the orientation of the epipolar lines and determine location of the epipoles for the resulting camera configurations.

$$t_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad t_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad t_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

- b. In the third case of the previous problem, where does the baseline lie and what does that imply for the location of the epipoles?

**Solutions:**

- a. The essential matrix  $E$  is given by  $E = [t]_{\times}R$ . Then with:

$$[t_i]_{\times} = \begin{bmatrix} 0 & -t_i^z & t_i^y \\ t_i^z & 0 & -t_i^x \\ -t_i^y & t_i^x & 0 \end{bmatrix}$$

We have:

$$\begin{aligned} E_{t_1} &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} I = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \\ E_{t_2} &= \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \\ E_{t_3} &= \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned}$$

Now that we have the essential matrices, we can reason about the corresponding epipolar lines. For  $t_1$ :

$$\tilde{l}_r = E_{t_1} \tilde{x}_l = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} x_l \\ y_l \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \\ y_l \end{pmatrix}, \quad \tilde{l}_l = E_{t_1}^T \tilde{x}_r = \begin{pmatrix} 0 \\ 1 \\ -y_r \end{pmatrix} \Rightarrow \text{Horizontal}$$

For  $t_2$ :

$$\tilde{l}_r = E_{t_2} \tilde{x}_l = \begin{pmatrix} 1 \\ 0 \\ -x_l \end{pmatrix}, \quad \tilde{l}_l = E_{t_2}^T \tilde{x}_r = \begin{pmatrix} -1 \\ 0 \\ x_r \end{pmatrix} \Rightarrow \text{Vertical}$$

For  $t_3$ :

$$\tilde{l}_r = E_{t_3} \tilde{x}_l = \begin{pmatrix} -y_l \\ x_l \\ 0 \end{pmatrix}, \quad \tilde{l}_l = E_{t_3}^T \tilde{x}_r = \begin{pmatrix} y_r \\ -x_r \\ 0 \end{pmatrix} \Rightarrow \text{Slanted}$$

From this we can conclude that for the first two cases, because all epipolar lines must intersect at the epipole and they are parallel, the epipoles must be ideal points located at infinity.

To determine the epipole in the third case, we choose two sets of values for  $x$  and  $y$  in the epipolar line equation for each image and find the intersection point of these two epipolar lines.

Consider

$$\tilde{l}_r = \begin{pmatrix} -y_l \\ x_l \\ 0 \end{pmatrix}$$

With  $x_l = 1; y_l = 1$  and  $x_l = 1; y_l = 2$ . Then we obtain the epipole as:

$$\tilde{e}_r = \begin{pmatrix} -1 \\ 1 \\ 0 \end{pmatrix} \times \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

The epipole for the first image can be determined analogously.

- b. Since we have pure translation in  $z$ -direction, the baseline lies on the optical/principal axis of the two cameras. Therefore the epipoles coincide with the principal point.

## FEATURES

In general, a feature is an attribute or property that represents an element. In the area of computer vision, features represent images or video properties that may be utilized to describe and understand the information. Corners, edges, angles, and colors are examples of **low-level features**, whereas items, scenes, and behaviors are examples of **high-level features**.

Features are crucial in computer vision because they facilitate the representation and analysis of visual input. Algorithms can find patterns and correlations in photos and videos by extracting and describing them with features and then making predictions or conclusions based on this knowledge.

Computer vision algorithms may reach great levels of accuracy and understanding of visual input by integrating low and high-level information.

A computer vision algorithm, for instance, may identify the existence of objects in an image by using low-level characteristics such as edges and corners. Next, the system might utilize high-level characteristics like objects and scenes to classify and identify the content or scenario of the image.

### Local Features

**Local features describe visual patterns or structures identifiable in small groups of pixels.** For example, edges, points, and various image patches.

They are essential aspects of an image or video that may be retrieved simply by applying straightforward techniques. Contours, edges, angles, and colors are instances of low-level characteristics. These features are frequently unique to a single image or video and have little relevance on their own.

### Edge detection

We want to convert a 2d image into a set of points where image intensity changes rapidly.

#### EDGE

We define edge as a rapid change in image intensity within small region.

#### *Causes of edges*

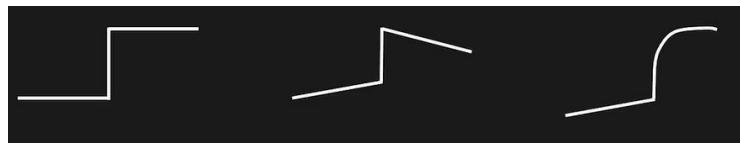
Rapid changes in image intensity are caused by various physical phenomena, like:

- Surface normal discontinuity (different surface orientation which gets different amount of light)
- Depth discontinuity (one object behind another)
- Surface reflectance discontinuity (writing upon the material)
- Illumination discontinuity (shadows)

#### *Types of edges*

There are different types of profiles for edges:

- Step edges



- Roof edge



- Line edges



We want an **Edge Operator** that produces:

- Edge position
- Edge magnitude (strength)
- Edge orientation (direction)

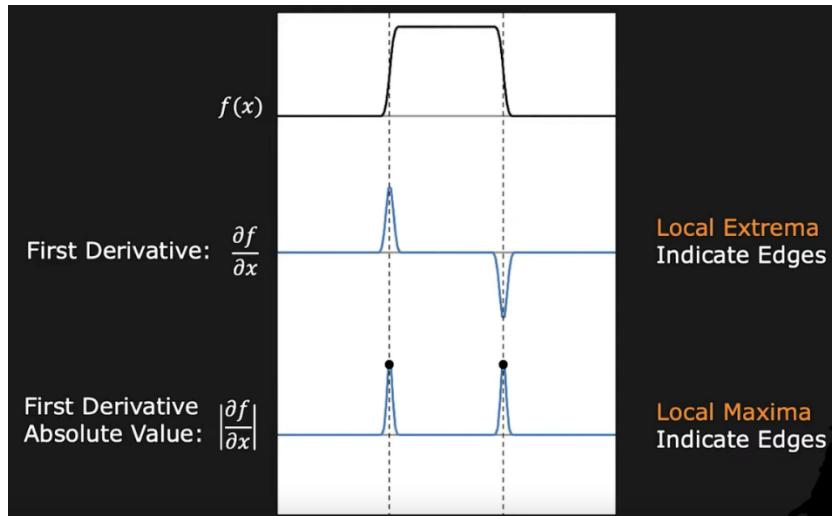
With the performance requirements:

- High detection rate
- Good localization
- Low noise sensitivity

## EDGE DETECTION USING GRADIENTS

### *1D Edge Detection using Derivative*

Edge is a rapid change in image intensity in a small region.



Using basic calculus, we know that, **derivative** of a continuous function represents the amount of change in the function.

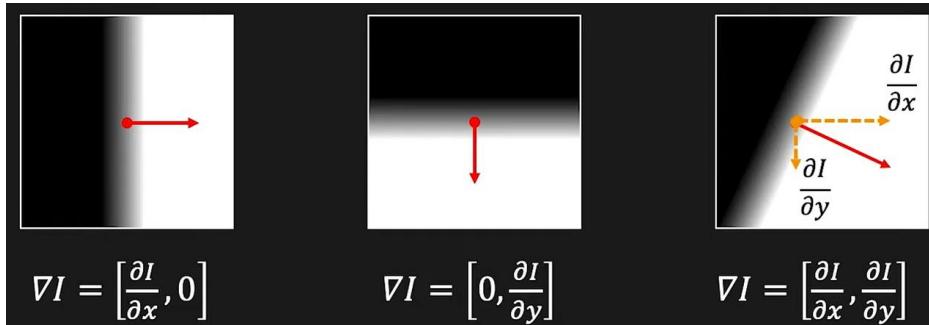
## 2D Edge Detection

Again, using basic calculus, **Partial Derivatives** of 2D continuous function represents the amount of change along each dimension.

Gradient

**Gradient (Partial Derivatives)** represents the direction of most rapid change in intensity.

$$\nabla I = \left[ \frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right]$$



**Gradient Magnitude:**

$$S = \|\nabla I\| = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}$$

**Gradient Orientation:**

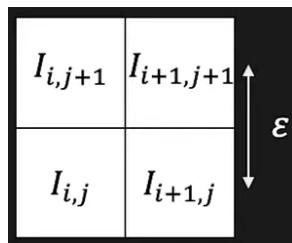
$$\theta = \arctan \left( \frac{\frac{\partial I}{\partial y}}{\frac{\partial I}{\partial x}} \right)$$

Discrete Gradient ( $\nabla$ ) Operator

Using **finite difference approximations**:

$$\begin{aligned} \frac{\partial I}{\partial x} &\approx \frac{1}{2\varepsilon} ((I_{i+1,j+1} - I_{i,j+1}) + (I_{i+1,j} - I_{i,j})) \\ \frac{\partial I}{\partial y} &\approx \frac{1}{2\varepsilon} ((I_{i+1,j+1} - I_{i+1,j}) + (I_{i,j+1} - I_{i,j})) \end{aligned}$$

Where:



This **can be implemented as convolution**:

$$\frac{\partial}{\partial x} \approx \frac{1}{2\varepsilon} \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$$

$$\frac{\partial}{\partial y} \approx \frac{1}{2\varepsilon} \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$$

Here the convolution flips have already been applied.

### Comparing Gradient ( $\nabla$ ) Operators

Gradient	Roberts	Prewitt	Sobel (3x3)	Sobel (5x5)
$\frac{\partial I}{\partial x}$	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & -2 & 0 & 2 & 1 \\ -2 & -3 & 0 & 3 & 2 \\ -3 & -5 & 0 & 5 & 3 \\ -2 & -3 & 0 & 3 & 2 \\ -1 & -2 & 0 & 2 & 1 \end{bmatrix}$
$\frac{\partial I}{\partial y}$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 3 & 5 & 3 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ -2 & -3 & -5 & -3 & -2 \\ -1 & -2 & -3 & -2 & -1 \end{bmatrix}$

For small operators we have:

- Good localization

But they are:

- Noise sensitive
- Poor detection

For large operators we have:

- Less noise sensitive
- Good detection

But:

- Poor localization

### Edge Thresholding

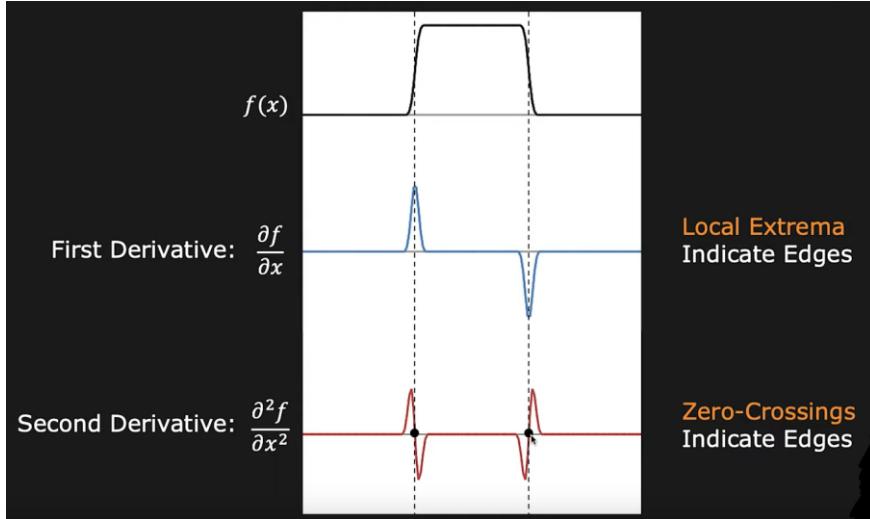
The **standard** way (Single Threshold  $T$ ):

$$\begin{cases} \|\nabla I(x, y)\| < T & \text{Definitely not an Edge} \\ \|\nabla I(x, y)\| \geq T & \text{Definitely an Edge} \end{cases}$$

Or using **Hysteresis Based** technique (Two Thresholds  $T_0 < T_1$ ):

$$\begin{cases} \|\nabla I(x, y)\| < T_0 & \text{Definitely not an Edge} \\ \|\nabla I(x, y)\| \geq T_1 & \text{Definitely an Edge} \\ T_0 \leq \|\nabla I(x, y)\| < T_1 & \text{Is an Edge if a Neighboring Pixel is Definitely an Edge} \end{cases}$$

## EDGE DETECTION USING LAPLACIAN



### Laplacian

The **Laplacian** is the sum of pure second derivatives:

$$\nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

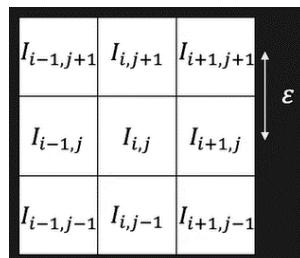
Where edges are “zero-crossings” in Laplacian of image, but we need to remember that the Laplacian doesn’t provide directions of edges.

Discrete Laplacian ( $\nabla^2$ ) Operator

Using **finite difference approximations**:

$$\begin{aligned} \frac{\partial^2 I}{\partial x^2} &\approx \frac{1}{\varepsilon} (I_{i-1,j} - 2I_{i,j} + I_{i+1,j}) \\ \frac{\partial^2 I}{\partial y^2} &\approx \frac{1}{\varepsilon} (I_{i,j-1} - 2I_{i,j} + I_{i,j+1}) \\ \nabla^2 I &= \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \end{aligned}$$

Where:

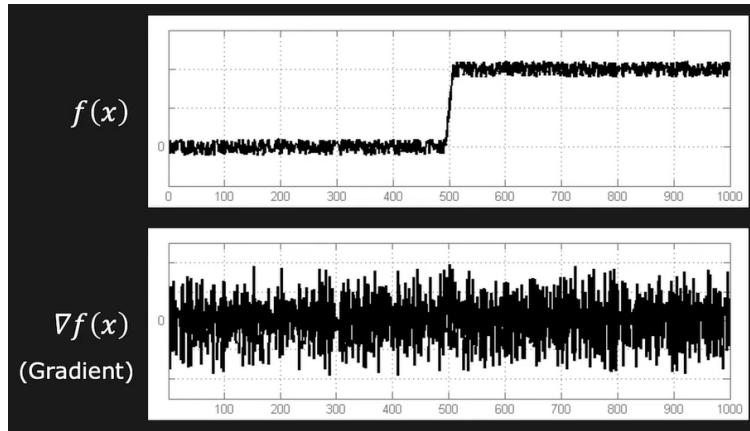


This can be implemented as convolution:

$$\nabla^2 \approx \frac{1}{\varepsilon^2} \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \text{ or } \nabla^2 \approx \frac{1}{6\varepsilon^2} \begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix}$$

Here the convolution flips have already been applied (second convolution is more accurate).

## EFFECTS OF NOISE



We need to suppress the noise before we apply edge detection, for example, using **Gaussian Smoothing**.

### *Derivative of Gaussian (DoG)*

In the Gaussian Smoothing we have used the derivative of Gaussian to find the edge:

$$\nabla(n_\sigma * f) = \nabla(n_\sigma) * f$$

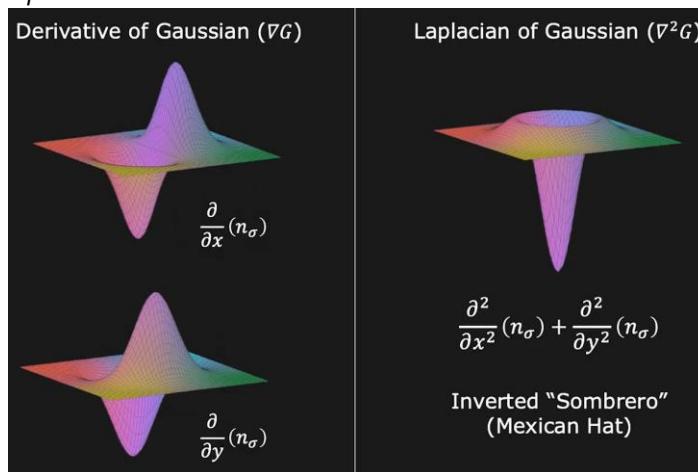
Here we used the fact that we know that the first derivative is a linear operation, and Gaussian smoothing is also linear.

### *Laplacian of Gaussian (LoG, $\nabla^2 n_\sigma$ , $\nabla^2 G$ )*

We can show the same for second derivative:

$$\nabla^2(n_\sigma * f) = \nabla^2(n_\sigma) * f$$

### *Gradient vs. Laplacian*



**The Gradient:**

- Provides location, magnitude and direction of the edge
- Detection using Maxima Thresholding
- Non-linear operation, requires two convolutions

**The Laplacian:**

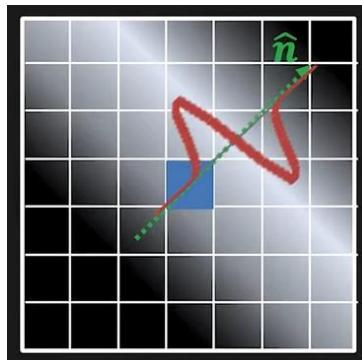
- Provides only the location of the edge
- Detection based on Zero-Crossing
- Linear operation, requires only one convolution

**CANNY EDGE DETECTOR**

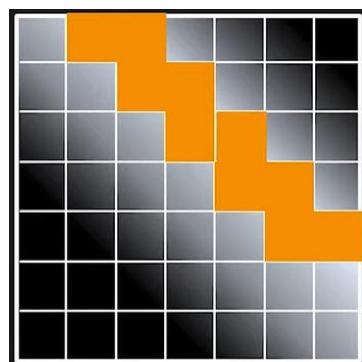
The **Canny edge detector** is the most widely used edge detector in computer vision, it uses the best properties of the Gradient operator and the Laplacian operator.

It is done using:

- Smooth image with 2D Gaussian:  $n_\sigma * I$
- Compute image gradient using **Sobel Operator**:  $\nabla n_\sigma * I$
- Find Gradient Magnitude at each pixel:  $\|\nabla n_\sigma * I\|$
- Find Gradient Orientation at each pixel:  $\hat{n} = \frac{\nabla n_\sigma * I}{\|\nabla n_\sigma * I\|}$
- Compute Laplacian along the Gradient Direction  $\hat{n}$  at each pixel:  $\frac{\partial^2(n_\sigma * I)}{\partial \hat{n}^2}$



- Find Zero Crossings in Laplacian to find the edge location



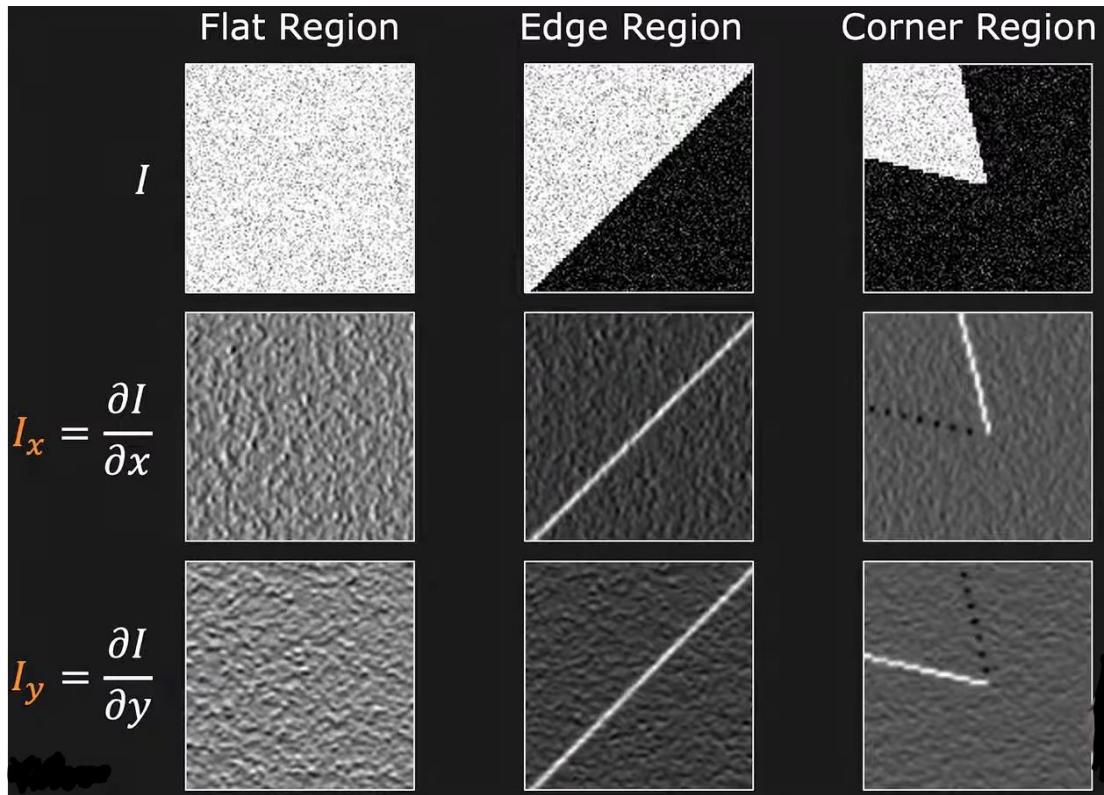
## Corner Detection

### CORNERS

We define **corner** as a point where two edges meet. i.e., rapid changes of image intensity in **two directions** within a small region.



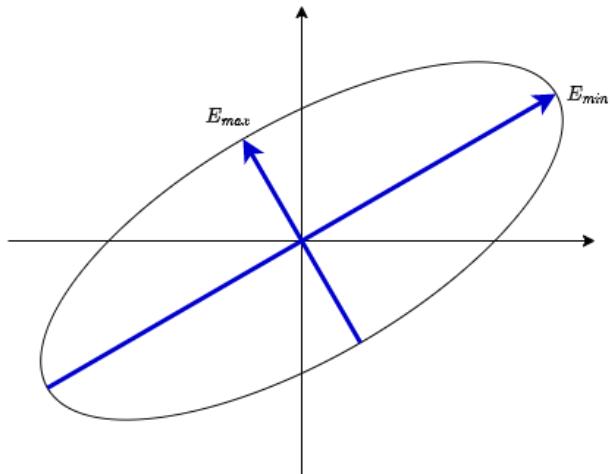
Using the image gradients, we get:



Where white is strong positive value ,black is strong negative value and grey is the average value (normalized).

We can classify the regions using the distribution of image gradients. We do it using fitting an Elliptical Disk to the distribution. Where we define:

- Maximum moment of inertia:  $E_{max}$
- Minimum moment of inertia:  $E_{min}$



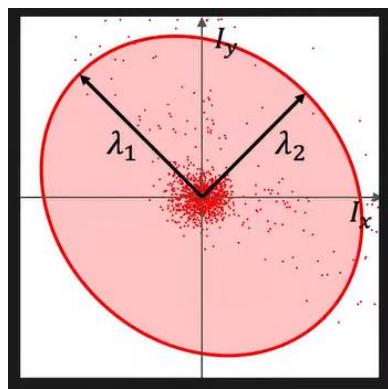
And:

- Length of Semi-Major Axis:  $\lambda_1 = E_{max}$
- Length of Semi-Minor Axis:  $\lambda_2 = E_{min}$

Second Moments for a region:

$$a = \sum_{i \in W} (I_{x_i}^2), \quad b = 2 \sum_{i \in W} (I_{x_i} I_{y_i}), \quad c = \sum_{i \in W} (I_{y_i}^2)$$

Where  $W$  is a window centered at pixel.



We get the **Ellipse Axes Lengths**:

$$\begin{aligned}\lambda_1 = E_{max} &= \frac{1}{2} [a + c + \sqrt{b^2 + (a - c)^2}] \\ \lambda_2 = E_{min} &= \frac{1}{2} [a + c - \sqrt{b^2 + (a - c)^2}]\end{aligned}$$

More generally, we can visualize  $H$  as an ellipse with **axis lengths** determined by the **eigenvalues** of  $H$  and **orientation** determined by the **eigenvectors** of  $H$ .

Solving:

$$E(u, v) \approx [u \ v] \underbrace{\begin{bmatrix} A & B \\ B & C \end{bmatrix}}_H \begin{bmatrix} u \\ v \end{bmatrix}$$

And getting:

$$\begin{aligned} Hx_{max} &= \lambda_{max}x_{max} \\ Hx_{min} &= \lambda_{min}x_{min} \end{aligned}$$

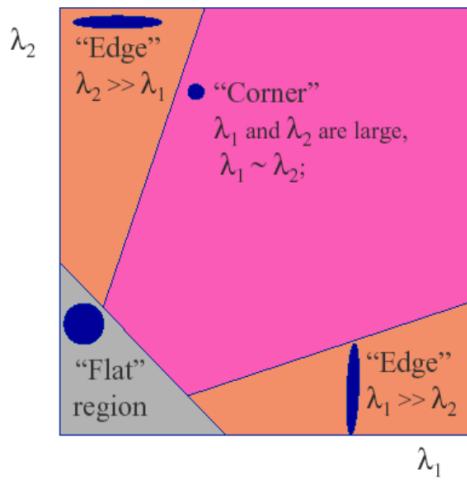
Eigenvalues  $\lambda$  and eigenvectors  $x$  of  $H$ :

- Define shift directions with the smallest and largest change in error
- $x_{max}$  is the direction of largest increase in  $E$
- $\lambda_{max}$  is the amount of increase in direction  $x_{max}$
- $x_{min}$  is the direction of smallest increase in  $E$
- $\lambda_{min}$  is the amount of increase in direction  $x_{min}$

Where:

- **Flat region:**  $\lambda_1 \approx \lambda_2$  (Both are small)
- **Edge region:**  $\lambda_1 \gg \lambda_2$  ( $\lambda_1$  is large and  $\lambda_2$  is small)
- **Corner region:**  $\lambda_1 \approx \lambda_2$  (Both are large)

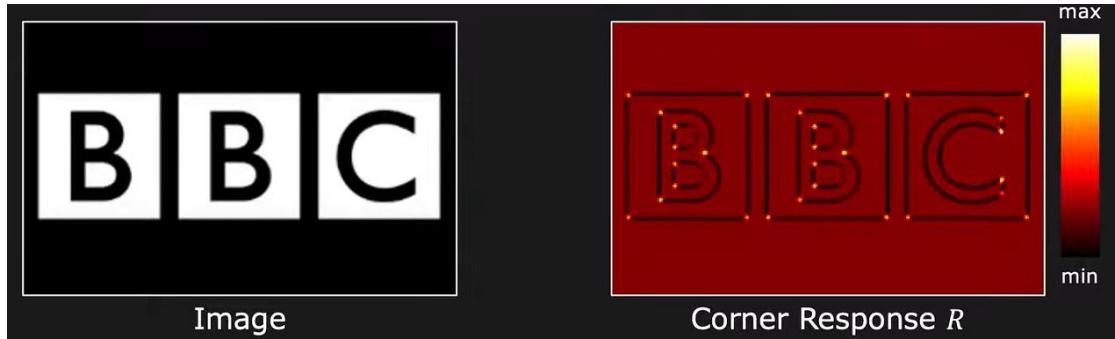
### HARRIS CORNER RESPONSE FUNCTION



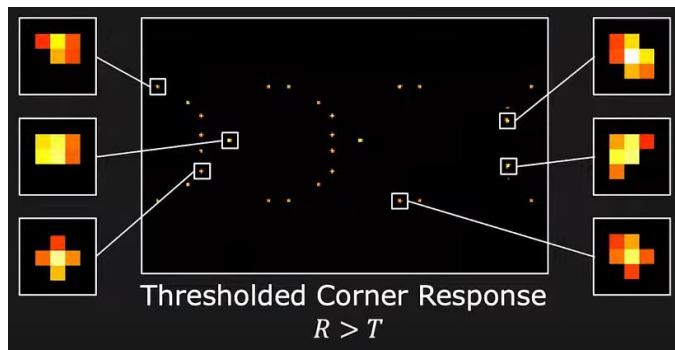
We can think about a classification problem in  $\lambda_1$  and  $\lambda_2$  space. Harris is come up with an expression (designed empirically):

$$R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2 = \det(H) - k \operatorname{tr}(H)^2, \quad 0.04 \leq k \leq 0.06$$

We can apply a threshold in  $R$  and using this threshold we can say if there is a corner or not.



But there is a problem around the corners, where we will get clusters of points that have large values, so we need to decide which one of them is the peak:



### *Non-Maximal Suppression (NMS)*

In this technique we:

1. Slide a window of size  $k$  over the image.
2. At each position, if the pixel at the center is the maximum value within the window, label it as positive (retain it). Else label it as negative (suppress it).



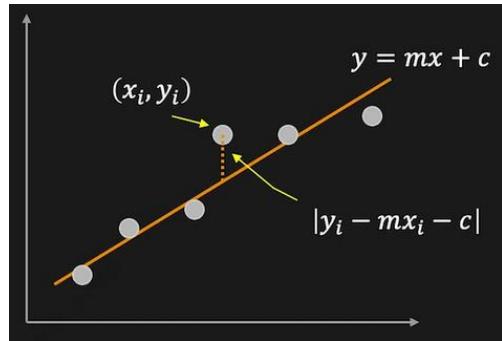
## Boundary Detection

We need to find **Object Boundaries** from **Edge Pixels**.

### FITTING LINES AND CURVES

Let's start talking about how we can fit a line to a set of edges. Given **Edge Points**  $(x_i, y_i)$ , we want to find a line that best represents the set of edges. we actually want to **minimize** the **average squared vertical distance**:

$$E = \frac{1}{N} \sum_i (y_i - mx_i - c)^2$$



We do so, using **Least Squares Solution**:

$$\frac{\partial E}{\partial m} = -2 \frac{1}{N} \sum_i x_i (y_i - mx_i - c) = 0, \quad \frac{\partial E}{\partial c} = \frac{-2}{N} \sum_i (y_i - mx_i - c) = 0$$

We get:

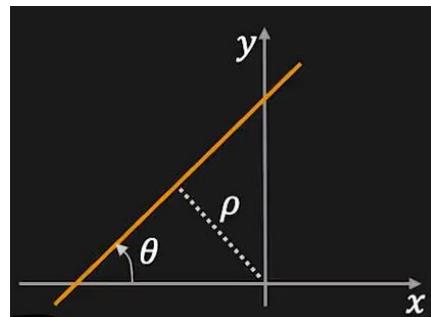
$$m = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2}, \quad c = \bar{y} - m\bar{x}$$

Where:

$$\bar{x} = \frac{1}{N} \sum_i x_i, \quad \bar{y} = \frac{1}{N} \sum_i y_i$$

But there is a problem when the points represent a vertical line. So we generalize and use a different line equation:

$$x \sin(\theta) - y \cos(\theta) + \rho = 0$$



Now we minimize the **average squared perpendicular distance**:

$$E = \frac{1}{N} \sum_i (x_i \sin(\theta) - y_i \cos(\theta) + \rho)^2$$

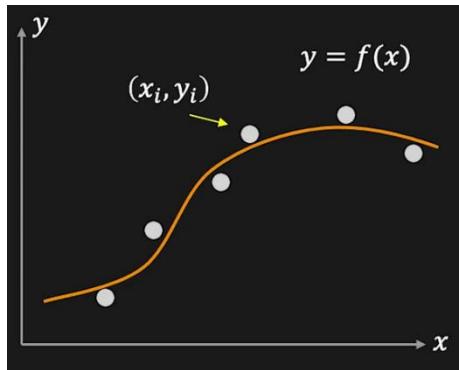
## FITTING CURVES TO EDGES

Now we are given **Edge Points**  $(x_i, y_i)$  and we want to find a **polynomial**:

$$y = f(x) = ax^3 + bx^2 + cx + d$$

That best represents the set of edges. we actually want to **minimize the average squared vertical distance**:

$$E = \frac{1}{N} \sum_i (y_i - ax_i^3 - bx_i^2 - cx_i - d)^2$$



We solve the **Linear System** using **Least Squares Fit** by:

$$\frac{\partial E}{\partial a} = 0, \quad \frac{\partial E}{\partial b} = 0, \quad \frac{\partial E}{\partial c} = 0, \quad \frac{\partial E}{\partial d} = 0$$

As we can see the closed-form solution cumbersome when unknowns are many.

$$\begin{cases} y_0 = ax_0^3 + bx_0^2 + cx_0 + d \\ y_1 = ax_1^3 + bx_1^2 + cx_1 + d \\ \vdots \\ y_i = ax_i^3 + bx_i^2 + cx_i + d \\ \vdots \\ y_n = ax_n^3 + bx_n^2 + cx_n + d \end{cases}$$

Given man  $(x_i, y_i)$ , this is an over-determined linear system with four unknowns  $(a, b, c, d)$ .

An over-determined linear system with  **$m$  unknowns**  $\{a_j\}_{j=0}^m$  and  **$n$  observations**

$\{(x_{ij}, y_i)\}_{j=0}^n$  ( $n > m$ ) can be written in a matrix form:

$$\begin{bmatrix} x_{00} & x_{01} & \dots & x_{0m} \\ x_{10} & x_{11} & \dots & x_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n0} & x_{n1} & \dots & x_{nm} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \Rightarrow Xa = y$$

But  $X_{n \times m}$  is not a square matrix and hence not invertible. But we can use **Least Squares**

**Solution:**

$$X^T X a = X^T y \Rightarrow a = (X^T X)^{-1} X^T y$$

Where  $X^+$  is the **Pseudo Inverse Matrix**:

$$X^+ = (X^T X)^{-1} X^T$$

So:

$$a = X^+ y$$

### ACTIVE CONTOURS (SNAKES)

Given an approximate boundary (contour) around the object, we want to evolve (move) the contour to fit exact object boundary.



Where the **Active Contour** iteratively “deform” the initial contour so that:

- It is near pixels with high gradient (edges)
- It is smooth

The **Deformable Contours** are boundaries that could deform over time (and be used to recognize the words that a person speaks):



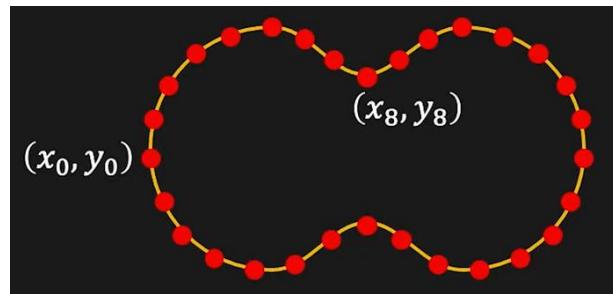
Or could deform with viewpoint:



**Boundary Tracking** means to use the boundary from the current image as initial boundary for the next image.

*Representing a Contour*

**Contour  $v$**  is defined as an ordered list of 2D vertices (control points) connected by **straight lines of fixed length**.



$$v = \{v_i = (x_i, y_i) | i = 0, 1, 2, \dots, n - 1\}$$

To move the contour to fit the object we want some “force” that will move to contour close to the object, we can use **Gradient Magnitude Squared** -  $\|\nabla I\|^2$ . Where we actually use **Blurred Gradient Magnitude Squared** -  $\|\nabla n_\sigma * I\|^2$ . We want to **maximize the sum of Gradient Magnitude Square**, which is same as **minimizing the negative of the sum of Gradient Magnitude Square**:

$$\text{Minimize : } E_{image} = - \sum_{i=0}^{n-1} \|\nabla n_\sigma * I(v_i)\|^2$$



## Contour Deformation: Greedy Algorithm

1. For each contour point  $\{v_i\}_{i=0}^{n-1}$ , move  $v_i$  to a position within a window  $W$  where the energy function  $E_{image}$  for the contour is minimum.
2. If the sum of the motions of all the contour points is less than a threshold, stop. Else go to step 1.



This greedy solution might be suboptimal and slow, and is sensitive to noise and initialization.

The solution is to add constraints that make the contour **contract and remain smooth**. We will **Minimize the Internal Bending Energy** of the contour:

$$E_{contour} = \alpha E_{elastic} + \beta E_{smooth}$$

Where  $(\alpha, \beta)$  control the influence of elasticity and smoothness.

*Elasticity and Smoothness*

For point  $0 \leq s \leq 1$  on continuous contour  $v(s) = (x(s), y(s))$ :

$$E_{elastic} = \left\| \frac{dv}{ds} \right\|^2, \quad E_{smooth} = \left\| \frac{d^2v}{ds^2} \right\|^2$$

Using discrete approximations at the control point  $v_i$ :

$$\begin{aligned} E_{elastic}(v_i) &= \left\| \frac{dv}{ds} \right\|^2 \approx \|v_{i+1} - v_i\|^2 = (x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 \\ E_{smooth} &= \left\| \frac{d^2v}{ds^2} \right\|^2 \approx \|(v_{i+1} - v_i) - (v_i - v_{i-1})\|^2 = \\ &= (x_{i+1} - 2x_i + x_{i-1})^2 + (y_{i+1} - 2y_i + y_{i-1})^2 \end{aligned}$$

Internal bending energy along the entire contour:

$$E_{contour} = \alpha E_{elastic} + \beta E_{smooth}$$

Where:

$$\begin{aligned} E_{elastic} &= \sum_{i=0}^{n-1} [(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2] \\ E_{smooth} &= \sum_{i=0}^{n-1} [(x_{i+1} - 2x_i + x_{i-1})^2 + (y_{i+1} - 2y_i + y_{i-1})^2] \end{aligned}$$

Combining the forces, where **Image Energy ( $E_{image}$ )** – measures how well the contour latches on to edges and **Internal Energy ( $E_{contour}$ )** – measures the elasticity and smoothness. The **Total Energy of Active Contour**:

$$E_{total} = E_{image} + E_{contour}$$

Contour Deformation: Greedy Algorithm

We can write a greedy algorithm again:

1. Uniformly sample the contour to get  $n$  contour points.
2. For each contour point  $\{v_i\}_{i=0}^{n-1}$ , move  $v_i$  to a position within a window  $W$  where the energy function  $E_{total}$  for the contour is minimum.

$$E_{total} = E_{image} + E_{contour}$$

3. If the sum of the motions of all the contour points is less than a threshold, stop. Else go to step 1.



Additional energy constraints can be added, for example:

- Penalize deviation from prior model or shape

But the technique requires good initialization:

- Edges cannot attract contours that are far away

As we have seen, elasticity makes contour contract (but we also can create contour inside the object):

- Replace contracting force with ballooning force to expand

## HOUGH TRANSFORM

One of the problems with boundary detection is knowing which edges in an image actually correspond to the boundary that we are looking for. The difficulties for the **fitting approach**:

- **Extraneous Data** – Which point to fit to?
- **Incomplete Data** – Only part of the model is visible (like some parts of the wheel which are incomplete).
- **Noise** (edges that are close to the wheel)

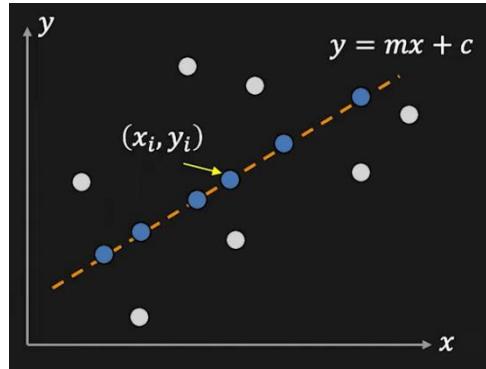


**Hough Transform** answers all those difficulties.

### *Line Detection*

As we did before, let's start talking about how we can fit a line to a set of edges. Given **Edge Points**  $(x_i, y_i)$ , we want to detect a line -  $y = mx + c$ . We consider point  $(x_i, y_i)$ , where the line is:

$$y_i = mx_i + c \Rightarrow c = -mx_i + y_i$$



We can look at the problem in two spaces:

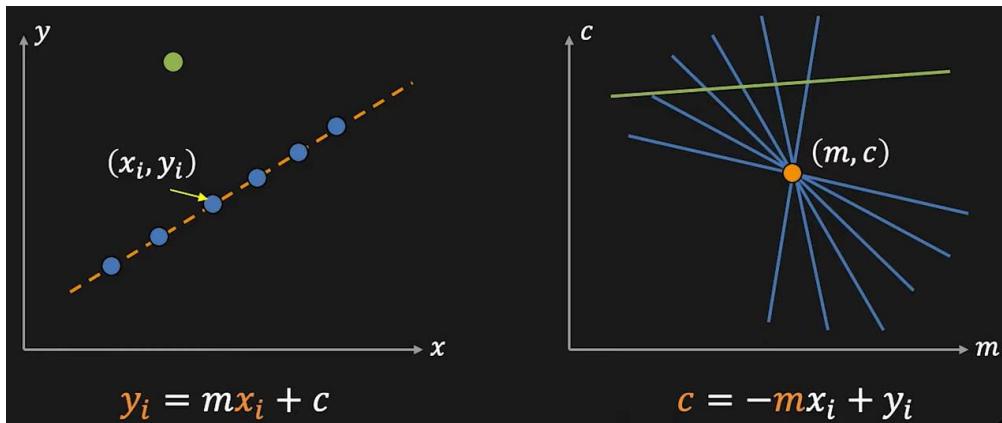
- Image Space (points locations  $(x_i, y_i)$  are known):

$$y_i = mx_i + c$$

- Parameters Space:

$$c = -mx_i + y_i$$

A point in image space gives a line in parameters space, if we have two (or more) points then they share only one intersection point  $(m, c)$ . A point that doesn't lie on the line from image space, then it doesn't intersect at point  $(m, c)$ .



Algorithm

1. Quantize parameter space  $(m, c)$
2. Create accumulator array  $A(m, c)$
3. Set  $A(m, c) = 0$  for all  $(m, c)$
4. For each edge point  $(x_i, y_i)$ :

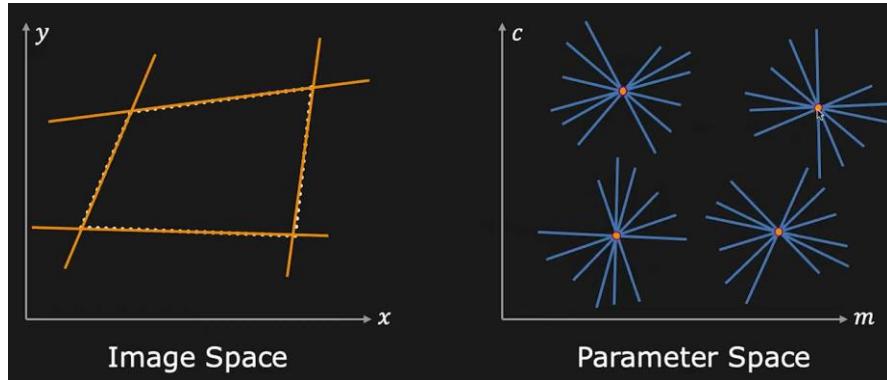
$$A(m, c) = A(m, c) + 1$$

If  $(m, c)$  lies on the line:  $c = mx_i + y_i$

5. Find local maxima in  $A(m, c)$

### Multiple Line Detection

Each line in image space is transformed to a point in parameters space:



### Better Parameterization

We have an issue that the slope of the line  $-\infty \leq m \leq \infty$ :

- Large Accumulator
- More Memory and Computation

The solution is to use the line as:  $x \sin(\theta) - y \cos(\theta) + \rho = 0$

- Orientation  $\theta$  is finite -  $0 \leq \theta \leq \pi$
- Distance  $\rho$  is finite

We look again at the problem in two spaces:

- Image Space (points locations are known):  

$$x_i \sin(\theta) - y_i \cos(\theta) + \rho = 0$$
- Parameters Space (parameters  $(\theta, \rho)$  are known):

$$x_i \sin(\theta) - y_i \cos(\theta) + \rho = 0$$

A point in image space gives a sinusoid line in parameters space, if we have two (or more) points then they share only two intersection points ( $\theta$  and  $\theta + \pi$ ). A point that doesn't lie on the line from image space, then it doesn't intersect at those points.

### Mechanics

#### How big should the accumulator be?

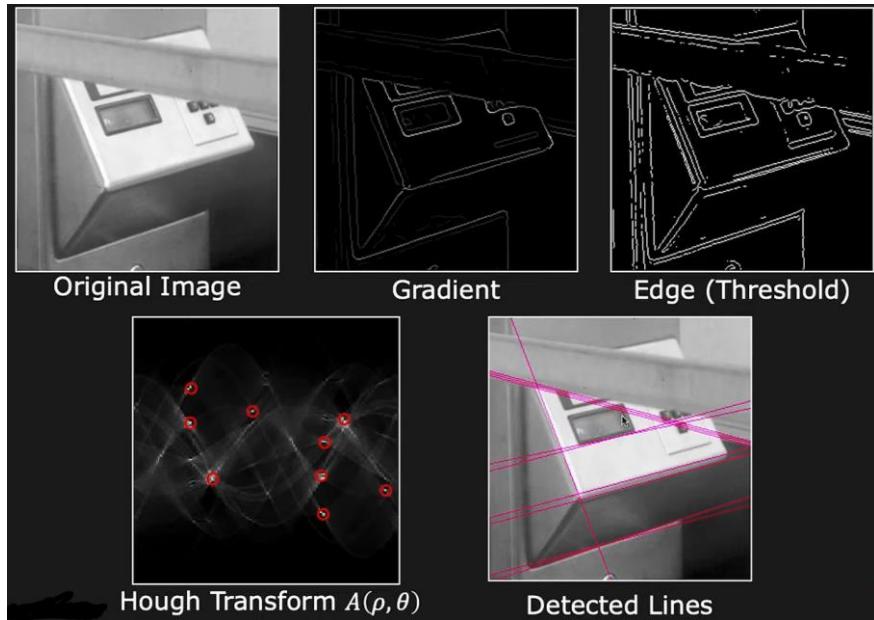
- Too big, and different lines may be merged
- Too small, and noise causes lines to be missed

#### How many lines?

- Count the peaks in the accumulator array (that are above some threshold)

#### Handling inaccurate edge locations:

- Increment patch in accumulator rather than single point (and use non-max suppression)



### *Circle Detection*

Equation of circle is given by:

$$(x_i - a)^2 + (y_i - b)^2 = r^2$$

If radius  $r$  is known, accumulate array -  $A(a, b)$ :

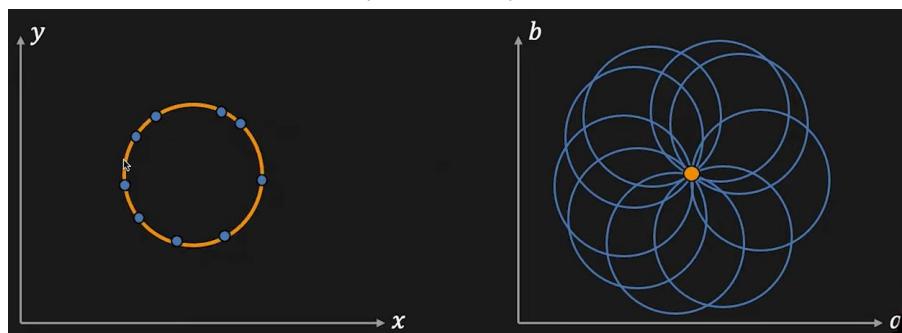
The two spaces:

- Image Space (points locations are known):

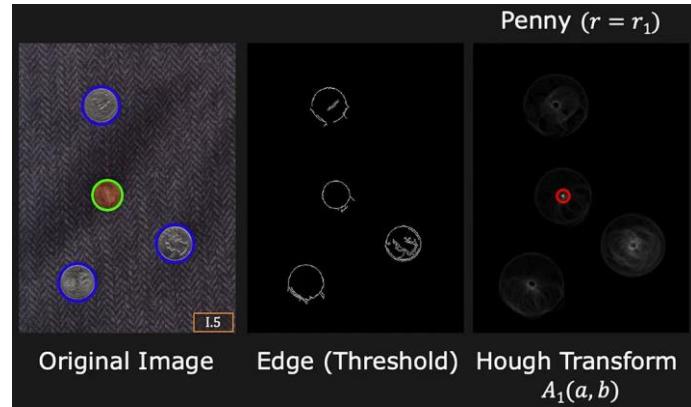
$$(x_i - a)^2 + (y_i - b)^2 = r^2$$

- Parameters Space:

$$(a - x_i)^2 + (b - y_i)^2 = r^2$$

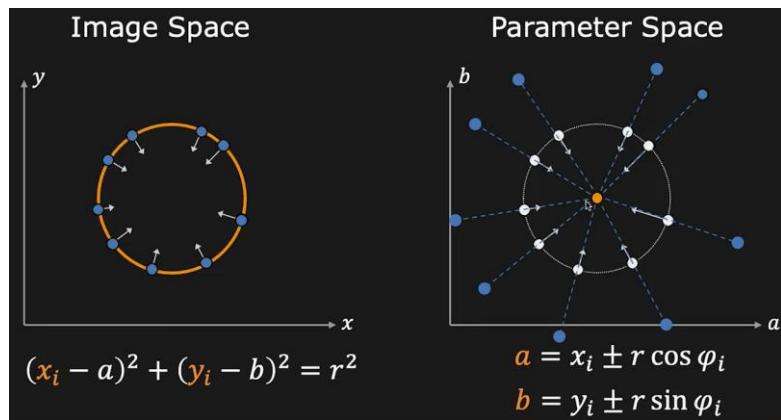


A point in image space gives a circle in parameters space, if we have three (or more) points then they share only one intersection point  $(m, c)$ . A point that doesn't lie on the line from image space, then it doesn't intersect at point  $(m, c)$ .



If we somehow given edge location  $(x_i, y_i)$ , know the edge direction  $\varphi_i$  and know the radius  $r$ .

For each point on the circle we know that the center of the circle lies on the orientation line of this point (at distance  $r$ ).



If radius  $r$  is not known – Accumulator Array -  $A(a, b, r)$

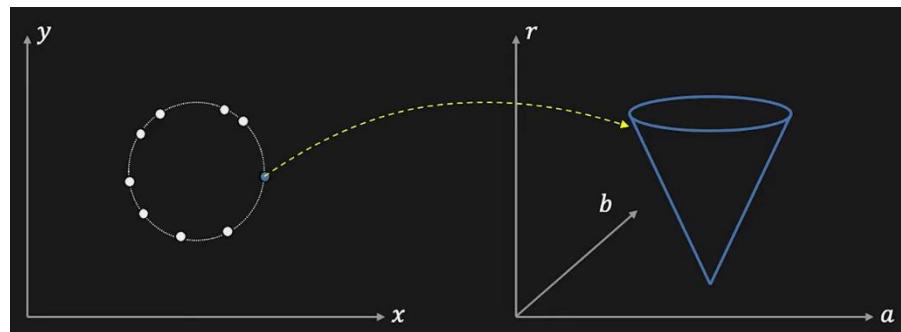
The two spaces:

- Image Space (points locations are known):

$$(x_i - a)^2 + (y_i - b)^2 = r^2$$

- Parameters Space (3D space):

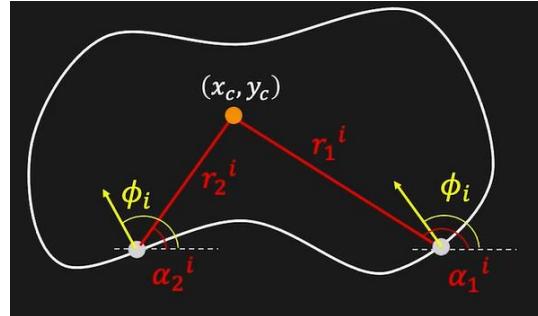
$$(a - x_i)^2 + (b - y_i)^2 = r^2$$



A point in image space gives a cone in parameters space, amount of work goes exponential larger if we don't know parameters.

## GENERALIZED HOUGH TRANSFORM

We want to find shapes that cannot be described by equations. We first define a **reference point**  $(x_c, y_c)$ , for each point on the objects' boundary we assume that we have **edge direction**  $\phi_i$ . We will represent each point using **edge location** -  $\vec{r}_k = (\vec{r}_k^i, \alpha_k^i)$ .



We create a  $\phi$ -Table:

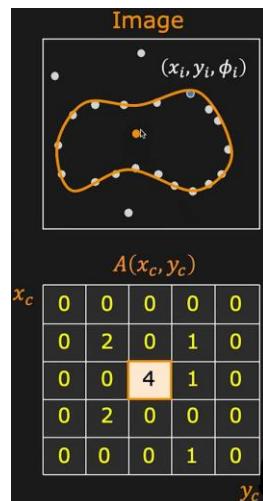
Edge Direction	$\vec{r} = (\vec{r}, \alpha)$
$\phi_1$	$\vec{r}_1^1, \vec{r}_2^1, \vec{r}_3^1$
$\phi_2$	$\vec{r}_1^2, \vec{r}_2^2$
$\vdots$	$\vdots$
$\phi_n$	$\vec{r}_1^n, \vec{r}_2^n, \vec{r}_3^n, \vec{r}_4^n$

### Algorithm

1. Create accumulator array  $A(x_c, y_c)$
2. Set  $A(x_c, y_c) = 0$  for all  $(x_c, y_c)$
3. For each edge point  $(x_i, y_i, \phi_i)$ :
  - o For each entry  $\phi_i \rightarrow r_k^i$  in  $\phi$ -table:
 
$$x_c = x_i \pm r_k^i \cos(\alpha_k^i)$$

$$y_c = y_i \pm r_k^i \sin(\alpha_k^i)$$

$$A(x_c, y_c) = A(x_c, y_c) + 1$$
4. Find local maxima in  $A(x_c, y_c)$



**The Hough Transform:**

- Works on disconnected edges
- Relatively insensitive to occlusion and noise
- Effective for simple shapes (lines, circles, etc.)
- Complex shapes: Generalized Hough Transform
- Trade-off between work in image space and parameters space

**EXAMPLE QUESTION – FITTING A CIRCLE**

We have discussed several methods for fitting a model. Now we are trying to fit a circle/circles on a 2D plane given a bunch of data points. Every data point is represented by  $(x_i, y_i)$  coordinates.

- a. How many parameters do you need to describe any circle in a 2D plane? At least how many data points do you need to fit a circle?
  - b. Suppose that you have enough points to fit a circle. First, let's consider the least squares method (it seems a good default choice). Write the linear system in the form of  $Ax = b$ , whose solution minimizes the objective function in a least squares sense, and explicitly explain what parameters you are fitting in your linear system.
- Hint:** The parameters you are fitting may not be exactly the same as parameters we usually use to describe a circle, but a substitution with an auxiliary variable  $z$  that is dependent on the circle parameters may be required.
- c. Another fitting method we've covered in class is the Hough transform. Let's define the Hough space according to the parameters you defined in (b). What does a data point in original space correspond to in this Hough space? Briefly describe how we can use the Hough transform to fit observations from a single circle.
  - d. The Hough transform can also be used to fit more than one circle. Suppose that you are given 10 data points. Five of them could perfectly fit one circle, and the other five perfectly fit another circle. Briefly describe what the Hough space looks like according to our definition in (c).
  - e. Someone gives you several data points and they all lie on a line. Unfortunately, this person still wants you to fit a circle (impossible!). If you have to bite the bullet and use the Hough space defined above to fit these points, what would the Hough space look like?

**Solutions:**

- a. Three. Three.
- b. Intuitively, we usually use  $(c_x, c_y, R)$  to describe a circle. For each  $(x_i, y_i)$ , we want the property:

$$(x_i - c_x)^2 + (y_i - c_y)^2 = R^2$$

which can be rewritten as:

$$2x_i c_x + 2y_i c_y + (R^2 - c_x^2 - c_y^2) = x_i^2 + y_i^2$$

So we can use least squares to fit parameters  $(c_x, c_y, R^2 - c_x^2 - c_y^2)$ . Note that we can always recover  $(c_x, c_y, R)$  from this set of parameters. So the linear system to solve least squares is:

$$\begin{pmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ \vdots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} c_x \\ c_y \\ R^2 - c_x^2 - c_y^2 \end{pmatrix} = \begin{pmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \vdots \end{pmatrix}$$

- c. A plane. These planes will intersect at a point, which defines the equation of the circle.
- d. There are 10 planes in the Hough space. Five of them intersect at one point, and the other five planes intersect at another point.
- e. Each pair of planes in the Hough space intersect at one line. All the intersection lines are parallel.

## SIFT Detector

A well-known and very robust algorithm for detecting interesting points and computing feature descriptions is **SIFT** which stands for **Scale-Invariant Feature Transform**.

### INTEREST POINT

We know that matching becomes easier if we can remove variations like size and orientation. So, an **Interesting Point/Feature** is one that:

- Has **rich image content** (brightness variation, color variation, etc.) within local window.
- Has well-defined **representation (signature)** for matching/comparing with other points.
- Has a well-defined **position** in the image.
- Should be **invariant to image rotation and scaling**.
- Should be **insensitive to light changes**.

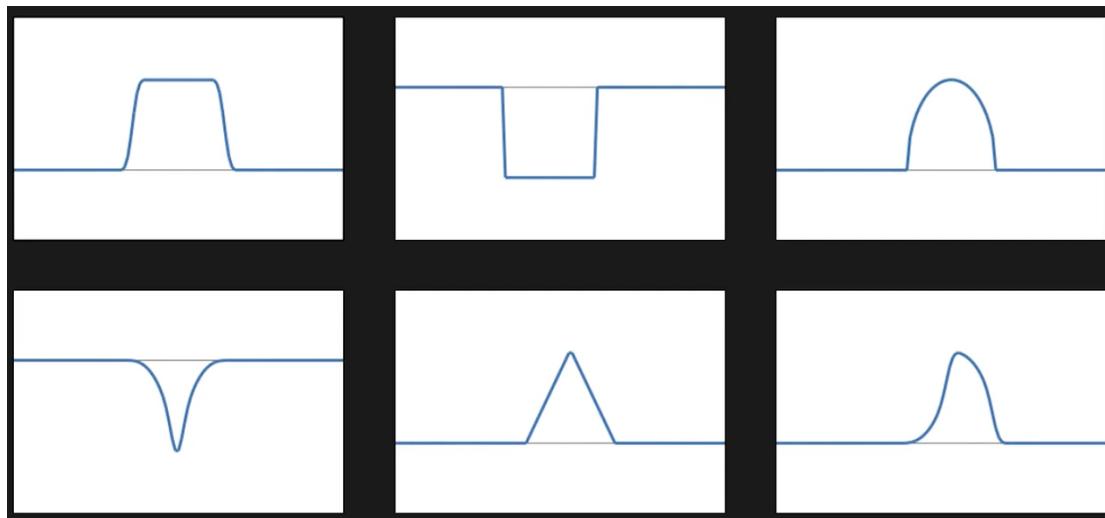
Lines/Edges aren't interesting enough, because they are not unique in image, it's **hard to localize** an edge and is sensitive to light. Corners have been used in the early stages of computer vision, but they also don't give enough information.

**Blobs** have **fixed position** and defined **size**. For a **Blob-like Feature** to be useful, we need to:

- **Locate** the blob
- Determine its **size**
- Determine its **orientation**
- Formulate a **description** or signature that is independent of size and orientation

### DETECTING BLOBS

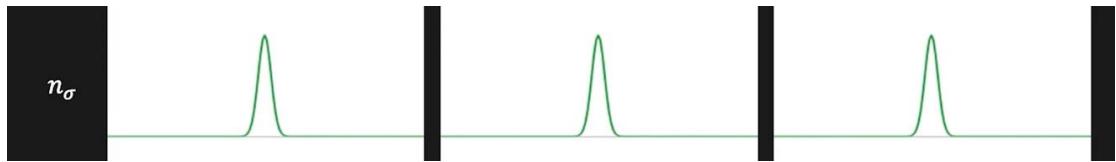
#### 1D Blobs



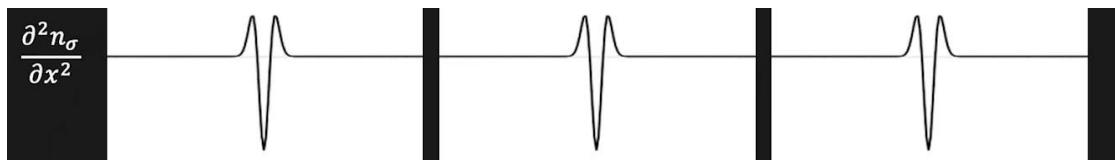
For example we will look at blobs that vary in size (*Blob A < Blob B < Blob C*):



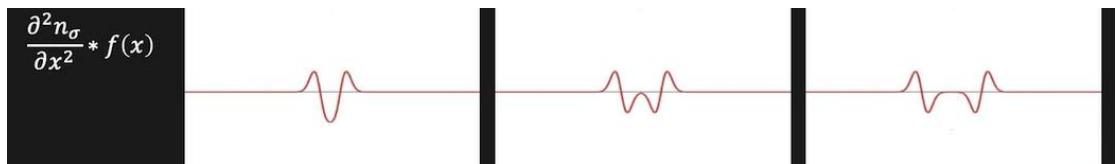
The **Gaussian**, where the  $\sigma$  is very important, it allow us to explore the scale space associated with the image:



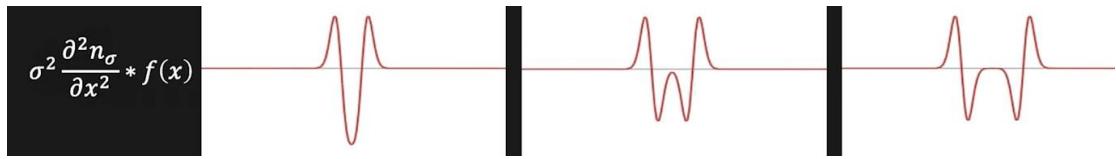
The second derivative of the Gaussian (LoG):



Applying LoG to the blobs we get:



For blob C we get two distinct zero-crossings, but for blob A and blob B, the zero-crossings began to overlap. To fix this problem we multiply the LoG with  $\sigma^2$ , as a result we get **sigma normalized output**:



We will apply it on multiple sigmas to get the center of the other blobs, where the width of Gaussian is proportional to the  $\sigma$  at which we found the center of the blob. The **Characteristic Scale** is the  $\sigma$  at which  $\sigma$ -normalized LoG attains its extreme value.

$$\text{Characteristic Scale} \propto \text{Size of Blob}$$

So given 1D signal  $f(x)$ , we want to compute  $\sigma^2 \frac{\partial^2 n_\sigma}{\partial x^2} * f(x)$  at many scales  $(\sigma_0, \sigma_1, \sigma_2, \dots, \sigma_k)$ .

And find:

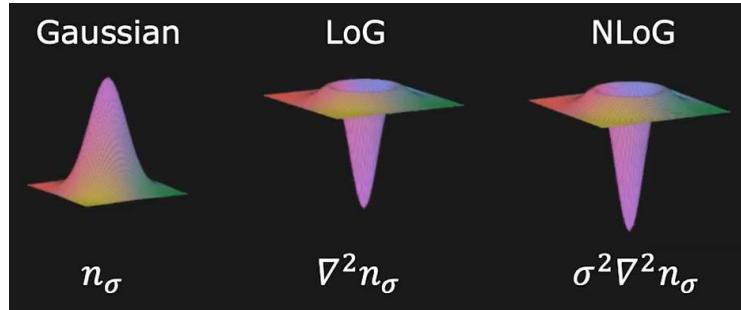
$$(x^*, \sigma^*) = \arg \max_{(x, \sigma)} \left| \sigma^2 \frac{\partial^2 n_\sigma}{\partial x^2} * f(x) \right|$$

Where:

- $x^*$  is the **Blob Position**
- $\sigma^*$  is the **Characteristic Scale** (blob size)

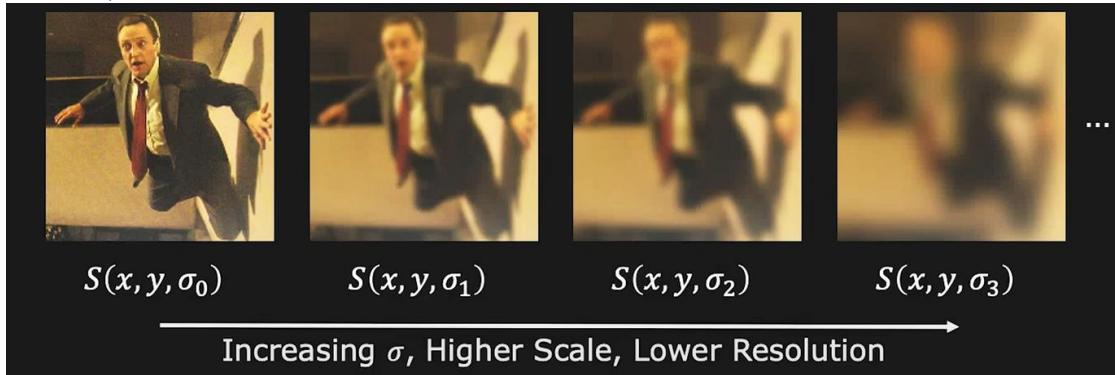
## 2D Blob Detector

**Normalized Laplacian of Gaussian (NLoG)** is used as the 2D equivalent for Blob Detection.



Location of blobs given by **Local Extrema** after applying Normalized Laplacian of Gaussian at many scales.

## Scale Space



Scale space is a stack created by filtering an image with Gaussians of different  $\sigma$ .

$$S(x, y, \sigma) = n(x, y, \sigma) * I(x, y)$$

Selecting sigmas to generate the scale-space:

$$\sigma_k = \sigma_0 s^k, \quad k = 0, 1, 2, \dots$$

Where:

- $s$  is a **Constant Multiplier**
- $\sigma_0$  is the **Initial Scale**

So given an image  $I(x, y)$ , we want to convolve the image using NLoG at many scales  $(\sigma_0, \sigma_1, \sigma_2, \dots, \sigma_k)$ . And find:

$$(x^*, y^*, \sigma^*) = \arg \max_{(x, y, \sigma)} |\sigma^2 \nabla^2 n_\sigma * I(x, y)|$$

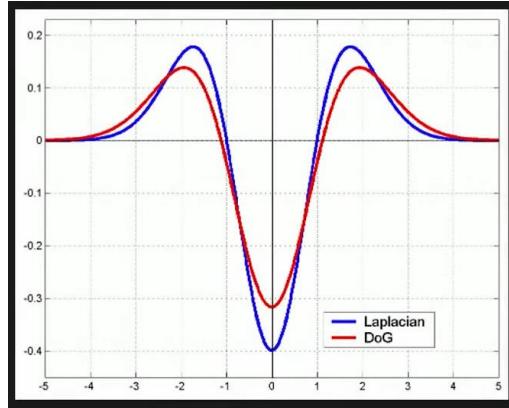
Where:

- $(x^*, y^*)$  is the **Blob Position**
- $\sigma^*$  is the **Characteristic Scale** (blob size)

## SIFT DETECTOR

A fast NLoG approximation is the **Difference of Gaussians** (DoG):

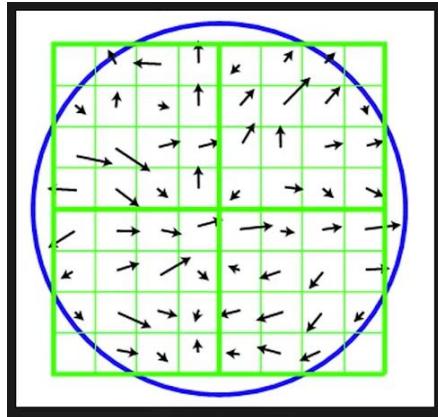
$$DoG = (n_{s\sigma} - n_\sigma) \approx (s - 1)\sigma^2 \nabla^2 n_\sigma$$



Rather than computing the NLoG, we can take stack of images (original image that is smoothed using different  $\sigma$ ) and find the difference between consecutive images, that is the NLoG (up to some scale).

At the second step we **find the extremum in every  $3 \times 3 \times 3$  grid**. From which we will get our interest point candidates. After moving weak extrema we have the final **SIFT interest points**.

### Computing the Principal Orientation

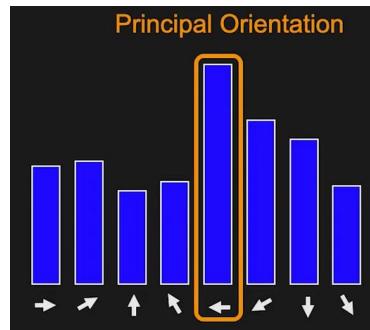


We look at the image gradient directions (inside the blob):

$$\theta = \arctan \left( \frac{\frac{\partial I}{\partial y}}{\frac{\partial I}{\partial x}} \right)$$

We will ignore the magnitude, as is affected by the lighting, gain of the camera, etc.

And we make a histogram of those directions:

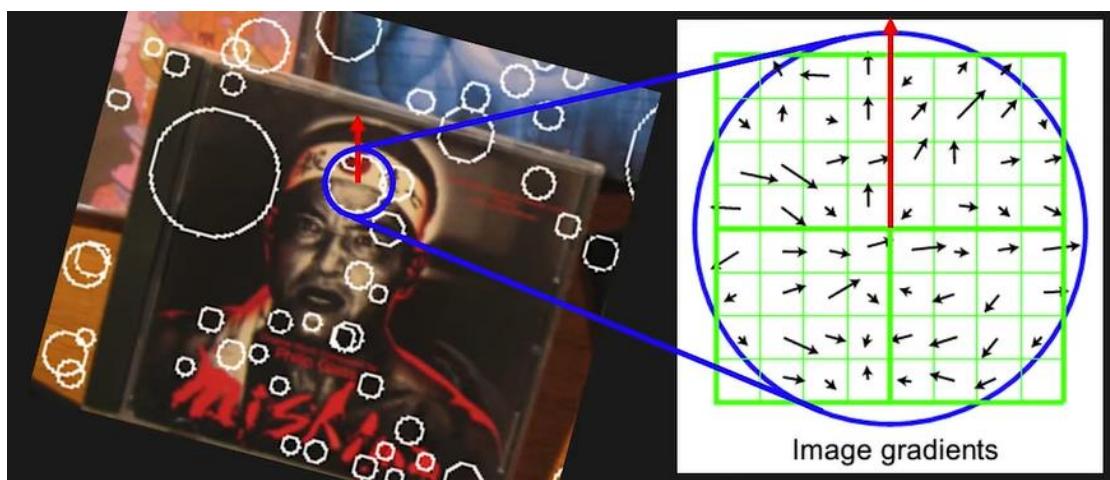


From which we choose the most prominent gradient direction, which is a **Principal Orientation**.

Using it we can undo the rotation.

### SIFT DESCRIPTOR

We want some signature that we can compute from a SIFT feature, so that we can match it with other features.



We can take the content (directions) of some feature and place it on a grid. We take the first quadrant from which we compute gradient orientation histogram, do this to all 4 quadrants and concatenate them all.

### Comparing SIFT Descriptors

Essentially comparing two arrays of data. Let  $H_1(k)$  and  $H_2(k)$  be two arrays of data of length  $N$ .

We can use  **$L_2$  Distance**:

$$d(H_1, H_2) = \sqrt{\sum_k (H_1(k) - H_2(k))^2}$$

Smaller the distance metric, better the match. Perfect match when  $d(H_1, H_2) = 0$ .

Or we can use **Normalized Correlation**:

$$d(H_1, H_2) = \frac{\sum_k [(H_1(k) - \bar{H}_1)(H_2(k) - \bar{H}_2)]}{\sqrt{\sum_k (H_1(k) - \bar{H}_1)^2} \sqrt{\sum_k (H_2(k) - \bar{H}_2)^2}}$$

Where:

$$\bar{H}_i = \frac{1}{N} \sum_{k=1}^N H_i(k)$$

Larger the distance metric, better the match. Perfect match when  $d(H_1, H_2) = 1$ .

Or, **Intersection**:

$$d(H_1, H_2) = \sum_k \min(H_1(k), H_2(k))$$

Where:

$$\bar{H}_i = \frac{1}{N} \sum_{k=1}^N H_i(k)$$

Larger the distance metric, better the match.

### *SIFT for 3D Objects*

SIFT is reliable for only small changes in viewpoint.

But still, it is Extraordinarily robust matching technique, because:

- Can handle changes in viewpoint (up to about 60 degrees out of plane rotation)
- Can handle significant changes in illumination (sometimes even day vs. night)
- Pretty fast to make real time, but can run in <1s for moderate image sizes
- Lots of code available

## High-Level Features

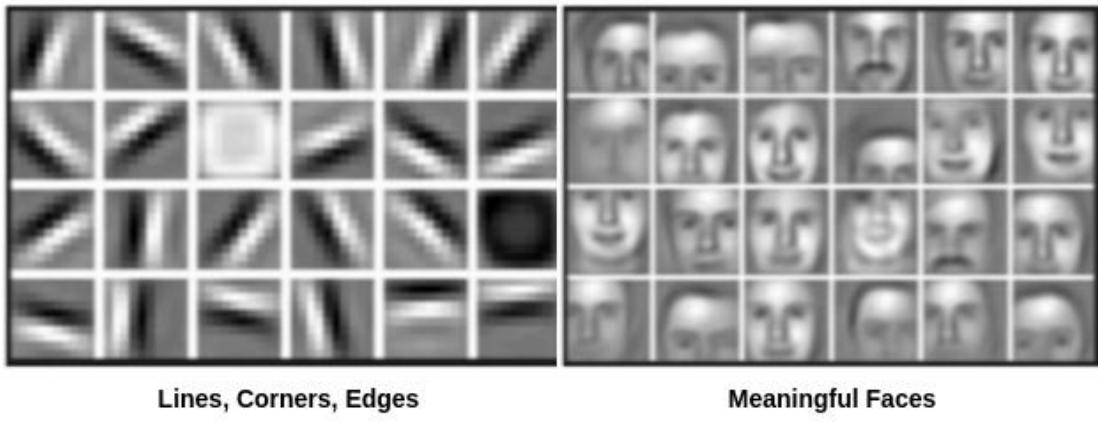
**High-level characteristics**, are more conceptual and thematically significant. They are **generated from low-level feature pairings and contain more complicated details about an image's or video's topic**. Items, scenarios, and interactions are instances of high-level features. These characteristics are more generic and may be applied to a broader variety of photos and videos.

Convolutional neural networks (CNNs) and recurrent neural networks (RNNs) are two major strategies for learning high-level features. These techniques may learn to extract and describe high-level features by being trained on big datasets of annotated images or videos.

Furthermore, pre-trained models that have previously been trained with large chunks of data are another technique for learning high-level characteristics. So, these models may be fine-tuned for special purposes and tasks and, therefore, can deliver accurate results with hardly any training.

## Difference between Low and High-Level Features

The main difference between low and high-level features lies in the fact that the first one is characteristics extracted from an image, such as colors, edges, and textures. In contrast, the second one is extracted from low-level features and denotes more semantically meaningful concepts. This basic difference can be easily understood by the diagram below:



### CONTENT

Furthermore, **low-level features** are more **closely related to the raw pixel data of the image and are more sensitive to noise and changes in the image**. In contrast, **high-level features** are more **robust and abstract and can be more helpful for tasks that require a higher level of understanding of the image content**. Also, high-level features have a clear semantic meaning and can be more easily interpreted by humans.

### SCALE

**Low-level attributes** are typically retrieved at a local scale, which means they are **vulnerable to little modifications of the picture**, like lighting or orientation. **High-level characteristics**, on the other hand, are frequently retrieved at a global scale, which means that **they take into account the whole picture or video and are more robust**.

### RESOURCES

Moreover, low-level feature extraction usually takes fewer system resources than high-level feature extraction, as the latter frequently requires more advanced machine learning methods.

### TASK SPECIFICITY

**Low-level characteristics** are frequently **task-specific**, which means they are appropriate for a certain set of activities. **High-level features**, on the other hand, are frequently **more generic and suitable for a broader range of jobs**. Lastly, **low-level features** are **useful for tasks such as image segmentation, object detection, and feature matching**. In contrast, **high-level features** are **useful for tasks such as image classification, object recognition, and scene understanding**.

## Descriptors

### TYPES OF FEATURE DESCRIPTORS

Depending on the application, we can take two different types of features from the photos. Both **local** and **global** elements describe them.

In general, we utilize **global features** for **low-level applications** like object detection and categorization, whereas we use **local features** for **higher-level applications** like object recognition. While increasing processing overheads, the combination of global and local features enhances recognition accuracy.

Between detection and identification, there is a significant distinction. Detection is finding anything or detecting an object's presence in the image. Recognition, on the other hand, is determining the identity of the detected object.

#### *Local descriptors*

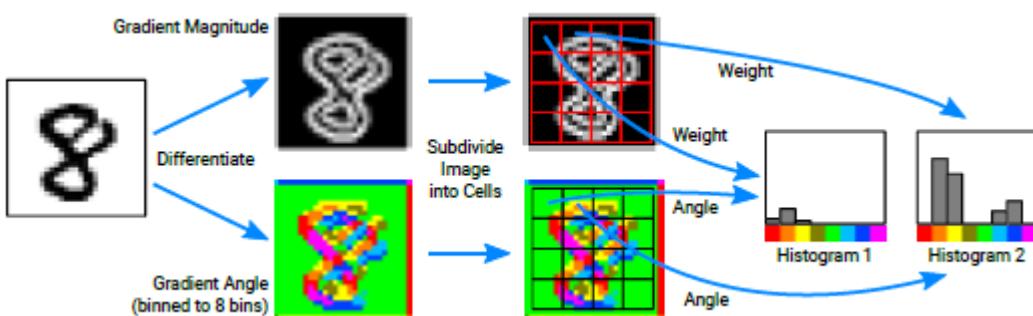
**Local features represent an image's texture.** A local descriptor describes a patch within an image. Using multiple local descriptors to match an image is more resilient than relying on one sole descriptor. The local descriptors SIFT, SURF, LBP, BRISK, MSER, and FREAK are a few examples.

#### *Global Descriptors*

**Local features describe the main areas inside an object's image, whereas global features describe the image as a whole to generalize the complete thing.** Contour representations, shape descriptors, and texture features are examples of global features. Global descriptors include things like shape matrices, invariant moments, histogram-oriented gradients (HOG), histograms of optical flow (HOF), and motion boundary histograms (MBH).

#### Histogram of Oriented Gradients (HoG)

Here we represent patches of the image with histograms of gradient angles weighted by the respective magnitude. Due to its invariance to small deformations such as translation, scale, rotation, and perspective, HoG has been the de facto standard for over 10 years.



## IMAGE STITCHING

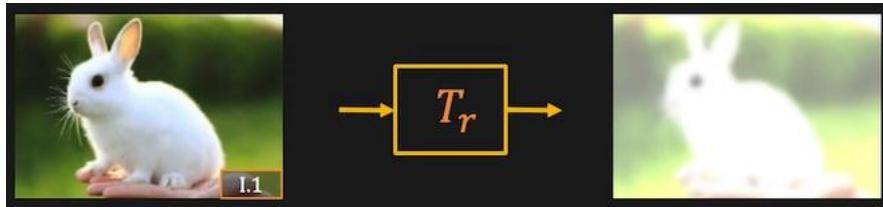
We have a 3D scene and we take a set of images of the scene from roughly the same viewpoint but by rotating the camera. We want to create a larger image (panorama).

## 2x2 Image Transformations

### IMAGE MANIPULATION

The most simple manipulation is **Image Filtering**, like the change of range (brightness) of the image:

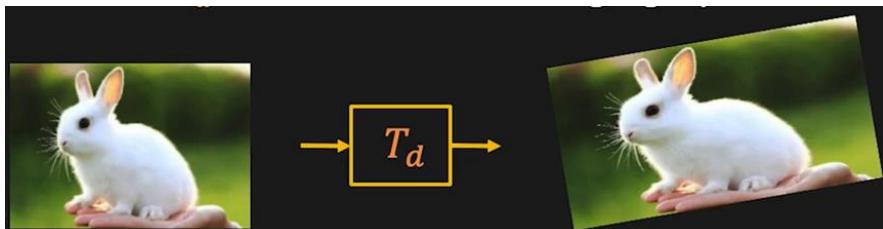
$$g(x, y) = T_r(f(x, y))$$



Another, is **Image Warping**, that is the change of the domain (location) of the image:

$$g(x, y) = T_d(f(x, y))$$

Where  $T_d$  is a coordinate changing operator.



### GLOBAL WARPING/TRANSFORMATION

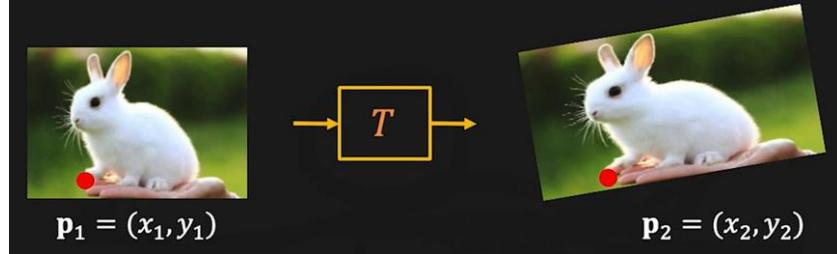
Transformations like:



$$g(x, y) = f(T(x, y))$$

Transformation  $T$  is the same over entire domain, often can be described by just a few parameters.

## 2X2 LINEAR TRANSFORMATIONS



$T$  can be represented by a matrix.

$$p_2 = T p_1 \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = T \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

*Scaling (Stretching or Squishing)*

**Forward:**

$$\begin{cases} x_2 = ax_1 \\ y_2 = by_1 \end{cases} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = S \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

**Inverse:**

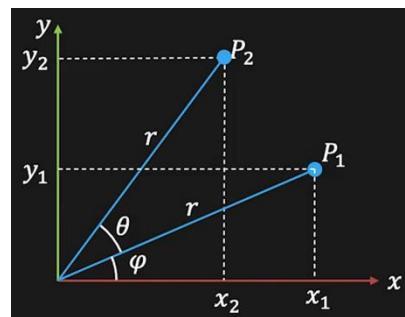
$$\begin{cases} x_1 = \frac{1}{a}x_2 \\ y_1 = \frac{1}{b}y_2 \end{cases} \Rightarrow \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = S^{-1} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} \Rightarrow \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \frac{1}{a} & 0 \\ 0 & \frac{1}{b} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$$

*2D Rotation*

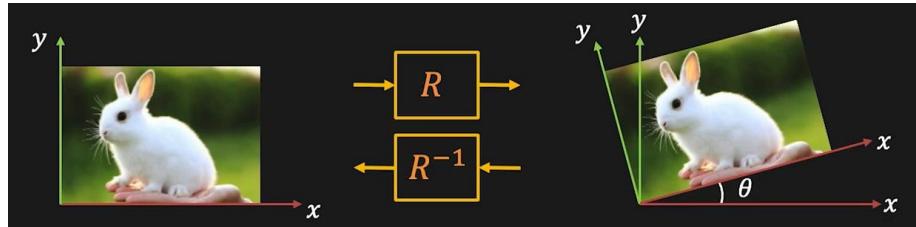
First we will represent the point in polar coordinates:

$$\begin{cases} x_1 = r \cos(\varphi) \\ y_1 = r \sin(\varphi) \end{cases}$$

Rotating by angle  $\theta$  (counter clock-wise):



$$\begin{cases} x_2 = r \cos(\varphi + \theta) \\ y_2 = r \sin(\varphi + \theta) \end{cases} \Rightarrow \begin{cases} x_2 = r \cos(\varphi) \cos(\theta) - r \sin(\varphi) \sin(\theta) \\ y_2 = r \cos(\varphi) \sin(\theta) + r \sin(\varphi) \cos(\theta) \end{cases} \Rightarrow \begin{cases} x_2 = x_1 \cos(\theta) - y_1 \sin(\theta) \\ y_2 = x_1 \sin(\theta) + y_1 \cos(\theta) \end{cases}$$



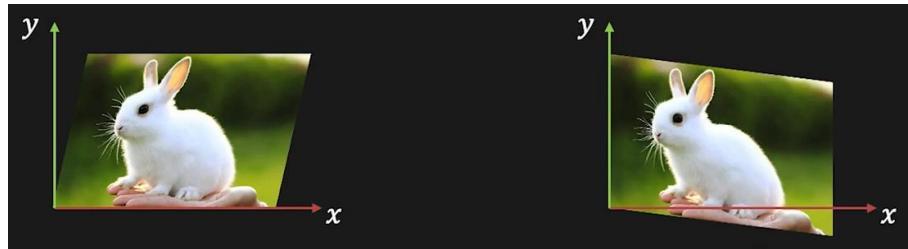
In matrix form. **Forward** (counter clockwise rotation):

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = R \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

**Inverse** (clockwise rotation):

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = R^{-1} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} \Rightarrow \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$$

*Skew*



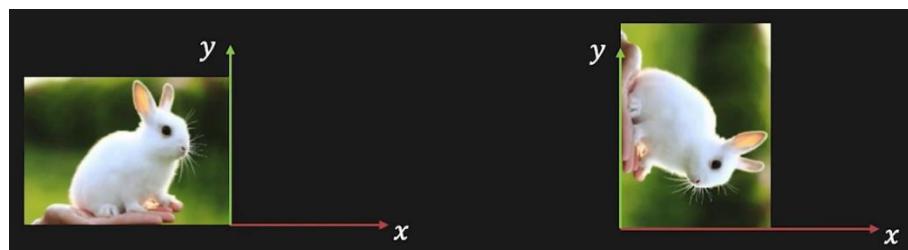
**Horizontal Skew:**

$$\begin{cases} x_2 = x_1 + m_x y_1 \\ y_2 = y_1 \end{cases} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = S_x \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & m_x \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

**Vertical Skew:**

$$\begin{cases} x_2 = x_1 \\ y_2 = m_y x_1 + y_1 \end{cases} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = S_y \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ m_y & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

*Mirror*



**Mirror about Y-axis:**

$$\begin{cases} x_2 = -x_1 \\ y_2 = y_1 \end{cases} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = M_y \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

**Mirror about line  $y = x$ :**

$$\begin{cases} x_2 = y_1 \\ y_2 = x_1 \end{cases} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = M_{xy} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

Properties of any Transformation of the form:

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

- Origin maps to the origin
- Lines map to lines
- Parallel lines remain parallel
- Closed under composition:

If  $p_2 = T_{21}p_1$  and  $p_3 = T_{32}p_2$  then there is  $p_3 = T_{31}p_1$ , such that:

$$p_3 = T_{32}p_2 = T_{32}(T_{21}p_1) \Rightarrow T_{31} = T_{32}T_{21}$$

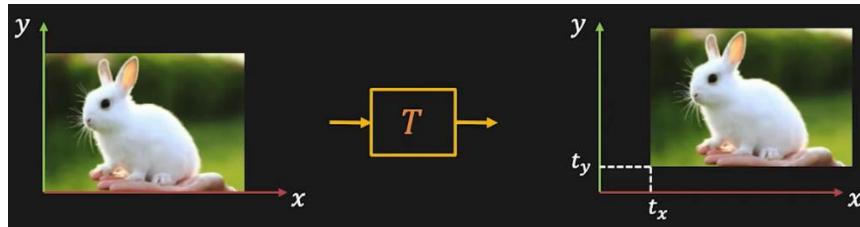
## 3x3 Image Transformations

Simple transformation like **translation** can't be expressed as a  $2 \times 2$  matrix. We will use

**Homogeneous Coordinates.** The **homogeneous** representation of a 2D point  $p = (x, y)$  is a 3D point  $\tilde{p} = (\tilde{x}, \tilde{y}, \tilde{z})$ . The third coordinate  $\tilde{z} \neq 0$  is **fictitious** such that:

$$\begin{cases} x = \frac{\tilde{x}}{\tilde{z}} \\ y = \frac{\tilde{y}}{\tilde{z}} \end{cases} \Rightarrow p = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \end{bmatrix} = \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \end{bmatrix} = \tilde{p}$$

Now we can easily show the **Translation**:



$$\begin{cases} x_2 = x_1 + t_x \\ y_2 = y_1 + t_y \end{cases} \Rightarrow \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

All previous transformations ( $2 \times 2$  image transformations), can be done using the  $3 \times 3$  matrix.

**Scaling:**

$$\begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

**Skew:**

$$\begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = \begin{bmatrix} 1 & m_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

**Translation** (2 DoFs):

$$\begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \Rightarrow \tilde{x}_2 = \begin{bmatrix} I & t \\ 0^T & 1 \end{bmatrix} \tilde{x}_1$$

**Rotation:**

$$\begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

**Euclidean** (Translation + Rotation: 3 DoFs):

$$\begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \Rightarrow \tilde{x}_2 = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} \tilde{x}_1$$

(in Euclidean transformation  $RR^T = I$  and  $\det(R) = 1$ ).

**Similarity** (Translation + Scaled Rotation: 4 DoFs):

$$\tilde{x}_2 = \begin{bmatrix} sR & t \\ 0^T & 1 \end{bmatrix} \tilde{x}_1$$

Composition of those transformations is just a multiplication in sequence. Those all are **Affine Transformations** (6 DoFs). Any transformation of the form:

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix} \begin{bmatrix} \tilde{x}_1 \\ \tilde{y}_1 \\ \tilde{z}_1 \end{bmatrix}$$

Properties of Affine Transformation:

- **Origin does not necessarily map to the origin**
- Lines map to lines
- Maps a line segment to a line segment
- Parallel lines (or segments) remain parallel
- Maps an  $n$ -gon to an  $n$ -gon
- Maps a parallelogram to a parallelogram
- Preserves the ratio of lengths of two parallel segments
- Preserves the ratio of areas of two figures
- Closed under composition

## PROOFS

- Let  $l$  be a line, and let  $l: \bar{p} + t\bar{u}$ ,  $t \in \mathbb{R}$ , be an equation of  $l$  in vector form. Then, for every  $t \in \mathbb{R}$ :

$$f(\bar{p} + t\bar{u}) = A(\bar{p} + t\bar{u}) + \bar{b} = A\bar{p} + At\bar{u} + \bar{b} = (A\bar{p} + \bar{b}) + t(A\bar{u}) = \bar{p}_1 + t\bar{u}_1$$

Where:

$$\bar{p}_1 = A\bar{p} + \bar{b}, \quad \bar{u}_1 = A\bar{u}$$

Hence:

$$f(l) = l_1$$

Where:

$$l_1: \bar{p}_1 + t\bar{u}_1, \quad t \in \mathbb{R}$$

Is again a line.

- If we restrict  $t$  to  $[0,1]$ , we get the proof for mapping line segments to line segments.
- Suppose that  $l: \bar{p} + t\bar{u}$ ,  $t \in \mathbb{R}$  and  $m: \bar{q} + t\bar{v}$ ,  $t \in \mathbb{R}$ , are parallel lines. Then  $\bar{v} = k\bar{u}$  for some  $k \in \mathbb{R}$ . Therefore:

$$f(\bar{p} + t\bar{u}) = A(\bar{p} + t\bar{u}) + \bar{b} = A\bar{p} + At\bar{u} + \bar{b} = (A\bar{p} + \bar{b}) + t(A\bar{u}) = \bar{p}_1 + t\bar{u}_1$$

And:

$$\begin{aligned} f(\bar{q} + t\bar{v}) &= A(\bar{q} + t\bar{v}) + \bar{b} = A\bar{q} + At\bar{v} + \bar{b} = (A\bar{q} + \bar{b}) + t(A\bar{v}) = \\ &= (A\bar{q} + \bar{b}) + t(A(k\bar{u})) = (A\bar{q} + \bar{b}) + tk(A\bar{u}) = \bar{q}_1 + tk\bar{u}_1 = \bar{q}_1 + t(k\bar{u}_1) \end{aligned}$$

That is,  $l$  and  $m$  are mapped to lines  $l_1$  and  $m_1$  that are parallel.

It is clear that for two line segments or a line and a line segment the proof is absolutely analogous.

- We prove this by strong induction on  $n$ . For the base case, when  $n = 3$ , consider a triangle  $T$ . Then  $T$  and its interior can be represented in vector form as  $T: \bar{u} + s\bar{v} + t\bar{w}$ , where  $s, t \in [0,1], s + t \leq 1$ , and the vectors  $\bar{v}$  and  $\bar{w}$  aren't collinear. Then:

$$\begin{aligned} f(T) &= f(\bar{u} + s\bar{v} + t\bar{w}) = A(\bar{u} + s\bar{v} + t\bar{w}) + \bar{b} = A\bar{u} + As\bar{v} + At\bar{w} + \bar{b} = \\ &= (A\bar{u} + \bar{b}) + s(A\bar{v}) + t(A\bar{w}) = \bar{u}_1 + s\bar{v}_1 + t\bar{w}_1 \end{aligned}$$

Where  $s, t \in [0,1], s + t \leq 1$ . By using the previous proof,  $\bar{v}_1 = A\bar{v}, \bar{w}_1 = A\bar{w}$  aren't parallel. Thus,  $T$  is mapped to a triangle  $T_1$ , which completes the proof of the base case.

Now suppose that  $f$  maps each  $n$ -gon to an  $n$ -gon for all  $n$ ,  $3 \leq n \leq k$ , and let  $P$  be a polygon with  $k + 1$  sides. We can prove that every polygon with at least 4 sides has a diagonal contained completely in its interior. Let  $\overline{AB}$  be such a diagonal in  $P$ . This diagonal divides  $P$  into two polygons,  $P_1$  and  $P_2$ , containing  $t$  and  $k + 3 - t$  sides, respectively, for some  $t, 3 \leq t \leq k$ . By the inductive hypothesis,  $f(P_1)$  and  $f(P_2)$  will be  $t$ -sided and  $(k + 3 - t)$ -sided polygons, respectively. Since each of these polygons will have the segment from  $f(A)$  to  $f(B)$  as a diagonal, the union of  $P_1$  and  $P_2$  will form a polygon with  $k + 1$  sides, which concludes the proof.

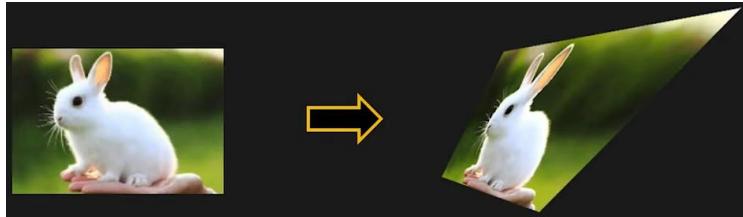
- The proof that a parallelogram is mapped to a parallelogram is analogous to the proof that triangles get mapped to triangles in the previous proof, by simply dropping the condition that  $s + t \leq 1$ .
- Consider parallel line segments,  $S_1$  and  $S_2$ , given in vector form as:

$$S_i: \bar{p}_i + t\bar{u}_i, \quad t \in [0,1]$$

Because they are parallel,  $\bar{u}_2 = k\bar{u}_1$  for some  $k \in \mathbb{R}$ . As  $|\bar{u}_i|$  is the length of  $S_i$ , the ratio of lengths of  $S_2$  and  $S_1$  is  $\frac{|k\bar{u}_1|}{|\bar{u}_1|} = |k|$ . From the previous proofs,  $S_i$  is mapped into

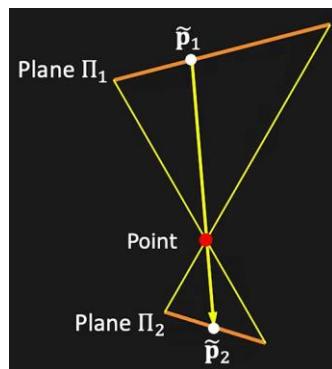
segment of length  $|A\bar{u}_i|$ . Since  $A\bar{u}_2 = A(k\bar{u}_1) = k(A\bar{u}_1)$ ,  $|A\bar{u}_2| = |k||A\bar{u}_1|$ . Which shows that the ratio of lengths of  $f(S_2)$  and  $f(S_1)$  is also  $|k|$ .

**Projective Transformation** on the other hand, is any transformation of the form:



$$\begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} \tilde{x}_1 \\ \tilde{y}_1 \\ \tilde{z}_1 \end{bmatrix} \Rightarrow \tilde{p}_2 = H\tilde{p}_1$$

This matrix is also called a **Homography** (8 DoFs). It is a mapping of one plane to another through a point.



It is same as imaging a plane through a pinhole (camera).

Homography can only be defined up to a scale:

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} \tilde{x}_1 \\ \tilde{y}_1 \\ \tilde{z}_1 \end{bmatrix} = \begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = k \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} \tilde{x}_1 \\ \tilde{y}_1 \\ \tilde{z}_1 \end{bmatrix}$$

If we fix scale such that -  $\sqrt{\sum(h_{ij})^2} = 1$  then 8 degrees of freedom. And hence the properties of the homography:

- **Origin does not necessarily map to the origin**
- Lines map to lines
- **Parallel lines do not necessarily remain parallel**
- Closed under composition

## TRANSFORMATION OF CO-VECTORS (E.G. LINES)

Considering any perspective 2D transformation, we can transform points using homogeneous transformations:

$$\tilde{x}_2 = \tilde{H}\tilde{x}_1$$

Using the 2D line equation, given by:

$$\tilde{l}_2^T \tilde{x}_2 = 0 \Rightarrow \tilde{l}_2^T (\tilde{H}\tilde{x}_1) = (\tilde{H}^T \tilde{l}_2)^T \tilde{x}_1 = 0$$

Then it must be:

$$(\tilde{H}^T \tilde{l}_2)^T \tilde{x}_1 = \tilde{l}_1^T \tilde{x}_1 = 0 \Rightarrow \tilde{l}_1 = \tilde{H}^T \tilde{l}_2$$

which we recognize as the line equation set to zero. Therefore, we have:

$$\tilde{l}_2 = (\tilde{H}^T)^{-1} \tilde{l}_1$$

Thus, the action of a **projective transformation on a co-vector** such as a 2D line or 3D normal can be represented by the transposed inverse of the matrix.

## SIMILARITY TRANSFORMATION

The matrix representation of a general **linear transformation is transformed from one frame to another** using a so-called **similarity transformation**.

Need to remember that rotational transformations do not commute.

### Example

Suppose a rotation matrix  $R$  represents:

- A rotation of angle  $\phi$  about the current y-axis followed by
- A rotation of angle  $\theta$  about the current z-axis

And suppose a rotation matrix  $R'$  represents:

- A rotation of angle  $\theta$  about the current z-axis followed by
- A rotation of angle  $\phi$  about the current y-axis

Where:

$$R_{y,\phi} = \begin{bmatrix} \cos \phi & 0 & \sin \phi \\ 0 & 1 & 0 \\ -\sin \phi & 0 & \cos \phi \end{bmatrix}, \quad R_{z,\theta} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Hence:

$$R = R_{y,\phi} R_{z,\theta} = \begin{bmatrix} \cos \phi & 0 & \sin \phi \\ 0 & 1 & 0 \\ -\sin \phi & 0 & \cos \phi \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos \phi \cos \theta & -\cos \phi \sin \theta & \sin \phi \\ \sin \theta & \cos \theta & 0 \\ -\sin \phi \cos \theta & \sin \phi \sin \theta & \cos \phi \end{bmatrix}$$

$$R' = R_{z,\theta} R_{y,\phi} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \phi & 0 & \sin \phi \\ 0 & 1 & 0 \\ -\sin \phi & 0 & \cos \phi \end{bmatrix} = \begin{bmatrix} \cos \phi \cos \theta & -\sin \theta & \sin \phi \cos \theta \\ \cos \phi \sin \theta & \cos \theta & \sin \phi \sin \theta \\ -\sin \phi & 0 & \cos \phi \end{bmatrix}$$

$$R \neq R'$$

*Example Questions*

Transformations

- a) Write down the  $2 \times 3$  translation matrix which maps  $(1,2)^T$  onto  $(0,3)^T$ .

Using a translation matrix:

$$\begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \Rightarrow \begin{pmatrix} 0 \\ 3 \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \Rightarrow$$

$$\Rightarrow \begin{cases} 0 = 1 + t_x \\ 3 = 2 + t_y \end{cases} \Rightarrow \begin{cases} t_x = -1 \\ t_y = 1 \end{cases} \Rightarrow T = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

- b) Let's assume that you are given  $N$  2D correspondence pairs:

$$(x_i, y_i) = \left( \begin{pmatrix} x_1^i \\ x_2^i \end{pmatrix}, \begin{pmatrix} y_1^i \\ y_2^i \end{pmatrix} \right)$$

Find the  $2 \times 3$  translation matrix mapping  $x_i$  onto  $y_i$  which is optimal in the least square sense.

**Hint:** Define a cost function as:

$$E(T) = \sum_{i=1}^N \|T\tilde{x}_i - y_i\|_2^2$$

and find the optimal  $T^*$  which minimizes  $E$ :

$$T^* = \arg \min_T E(T)$$

You can find  $T^*$  by calculating the Jacobian  $J_E$  of  $E$  and setting it to  $0^T$ :

$$J_E = \left[ \frac{\partial E}{\partial t_1}, \dots, \frac{\partial E}{\partial t_N} \right] = 0^T$$

Can you give an intuitive explanation for the equation you derive for  $T^*$ ?

We first observe:

$$T = \begin{bmatrix} 1 & 0 & t_1 \\ 0 & 1 & t_2 \end{bmatrix}$$

Next:

$$E(T) = \sum_{i=1}^N \|T\tilde{x}_i - y_i\|_2^2$$

$$E(t_1, t_2) = \sum_{i=1}^N (x_1^i + t_1 - y_1^i)^2 + (x_2^i + t_2 - y_2^i)^2$$

We then calculate the partial derivatives set them equal to 0:

$$\frac{\partial E}{\partial t_1} = \frac{\partial}{\partial t_1} \left( \sum_{i=1}^N (x_1^i + t_1 - y_1^i)^2 + (x_2^i + t_2 - y_2^i)^2 \right) = 2 \sum_{i=1}^N (x_1^i + t_1 - y_1^i) =$$

$$= 2 \sum_{i=1}^N (x_1^i - y_1^i) + 2 \sum_{i=1}^N t_1 = 2 \sum_{i=1}^N (x_1^i - y_1^i) + 2Nt_1 = 0$$

Rearranging gives:

$$t_1 = -\frac{2 \sum_{i=1}^N (x_1^i - y_1^i)}{2N} = \frac{1}{N} \sum_{i=1}^N (y_1^i - x_1^i)$$

And similarly, we obtain:

$$t_2 = \frac{1}{N} \sum_{i=1}^N (y_2^i - x_2^i)$$

We see that the optimal translation is the mean of the individual translations.

- c) You are given the following three correspondence pairs:

$$\begin{pmatrix} (0) \\ (1) \end{pmatrix}, \begin{pmatrix} (3) \\ (-5) \end{pmatrix}$$

$$\begin{pmatrix} (5) \\ (7) \end{pmatrix}, \begin{pmatrix} (7) \\ (6) \end{pmatrix}$$

$$\begin{pmatrix} (4) \\ (1) \end{pmatrix}, \begin{pmatrix} (5) \\ (-4) \end{pmatrix}$$

Using your derived equation, calculate the optimal  $2 \times 3$  translation matrix  $T^*$ .

Calculating the  $t$  vectors (as we did in (a)):

$$t_1 = (3, -6)^T$$

$$t_2 = (2, -1)^T$$

$$t_3 = (1, -5)^T$$

The optimal is:

$$t^* = \frac{1}{3}(t_1 + t_2 + t_3) = (2, -4)^T \Rightarrow T^* = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & -4 \end{bmatrix}$$

Transformation of a line

A 2D line can be expressed as  $ax + by + c = 0$  or in homogeneous terms:

$$l\tilde{x} = 0$$

Where  $l = (a, b, c)^T$ . If points are transformed so that:

$$\tilde{x}' = T\tilde{x}$$

what is the equation of the transformed line?

**Solution:**

It must be true that for the new line:

$$l'\tilde{x}' = 0$$

And so:

$$l'T\tilde{x} = 0$$

By inspection we can see that to make this true we can set:

$$l = l'T \Rightarrow l' = lT^{-1}$$

### Homography

A camera with intrinsic matrix  $\Lambda$  and extrinsic parameters  $R = I, t = 0$  takes an image and then rotates to a new position  $R = R_r, t = 0$  and takes a second image. Show that the homography relating these two images is given by:

$$\Phi = \Lambda R_r \Lambda^{-1}$$

#### Solution:

For the first image, we have:

$$\lambda_1 \tilde{x}_1 = \Lambda [I|0] \tilde{w} = \Lambda w$$

Similarly, for the second image we have:

$$\lambda_2 \tilde{x}_2 = \Lambda [R_r|0] \tilde{w} = \Lambda R_r w$$

From the first image we get the relation:

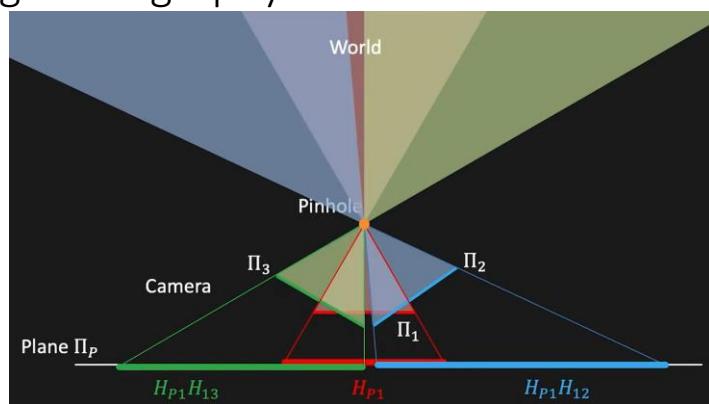
$$w = \lambda_1 \Lambda^{-1} x_1$$

We now substitute into the relation in the second camera to get:

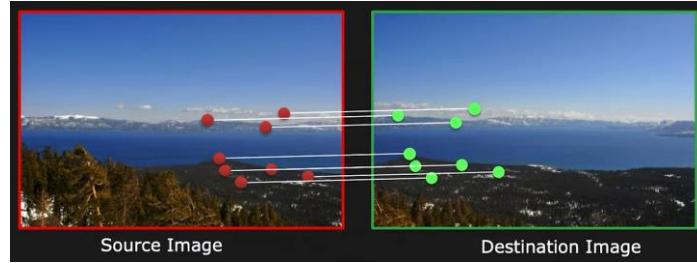
$$\lambda_2 \tilde{x}_2 = \Lambda [R_r|0] \tilde{w} = \Lambda R_r w = \Lambda R_r (\lambda_1 \Lambda^{-1} x_1) \Rightarrow \lambda_2 x_2 = \Lambda R_r \Lambda^{-1} x_1$$

as required.

### Computing Homography



We want given a set of matching features/points between images 1 and 2, to find the **homography  $H$**  that best “agrees” with the matches. The scene points should lie on a plane, or be distant (plane at infinity), or imaged from the same point. We first define a **Source Image** and **Destination Image**.



Such that:

$$\begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = \begin{bmatrix} \tilde{x}_d \\ \tilde{y}_d \\ \tilde{z}_d \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix}$$

It looks like we have 9 unknowns here, but as we said before there are actually 8 degrees of freedom, so we **need minimum of 4 matching points** (use more if you can, for robustness).

For a given pair  $i$  of corresponding points, we can write:

$$\begin{cases} x_d^{(i)} = \frac{\tilde{x}_d^{(i)}}{\tilde{z}_d^{(i)}} = \frac{h_{11}\tilde{x}_s^{(i)} + h_{12}\tilde{y}_s^{(i)} + h_{13}}{h_{31}\tilde{x}_s^{(i)} + h_{32}\tilde{y}_s^{(i)} + h_{33}} \\ y_d^{(i)} = \frac{\tilde{y}_d^{(i)}}{\tilde{z}_d^{(i)}} = \frac{h_{21}\tilde{x}_s^{(i)} + h_{22}\tilde{y}_s^{(i)} + h_{23}}{h_{31}\tilde{x}_s^{(i)} + h_{32}\tilde{y}_s^{(i)} + h_{33}} \end{cases} \Rightarrow$$

Rearranging the terms:

$$\Rightarrow \begin{cases} x_d^{(i)}(h_{31}\tilde{x}_s^{(i)} + h_{32}\tilde{y}_s^{(i)} + h_{33}) = h_{11}\tilde{x}_s^{(i)} + h_{12}\tilde{y}_s^{(i)} + h_{13} \\ y_d^{(i)}(h_{31}\tilde{x}_s^{(i)} + h_{32}\tilde{y}_s^{(i)} + h_{33}) = h_{21}\tilde{x}_s^{(i)} + h_{22}\tilde{y}_s^{(i)} + h_{23} \end{cases} \Rightarrow$$

Writing as linear equation:

$$\Rightarrow \begin{bmatrix} x_s^{(i)} & y_s^{(i)} & 1 & 0 & 0 & 0 & -x_d^{(i)}x_s^{(i)} & -x_d^{(i)}y_s^{(i)} & -x_d^{(i)} \\ 0 & 0 & 0 & x_s^{(i)} & y_s^{(i)} & 1 & -y_d^{(i)}x_s^{(i)} & -y_d^{(i)}y_s^{(i)} & -y_d^{(i)} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Combining the equations for all corresponding points:

$$\begin{bmatrix} x_s^{(1)} & y_s^{(1)} & 1 & 0 & 0 & 0 & -x_d^{(1)}x_s^{(1)} & -x_d^{(1)}y_s^{(1)} & -x_d^{(1)} \\ 0 & 0 & 0 & x_s^{(1)} & y_s^{(1)} & 1 & -y_d^{(1)}x_s^{(1)} & -y_d^{(1)}y_s^{(1)} & -y_d^{(1)} \\ \vdots & \vdots \\ x_s^{(i)} & y_s^{(i)} & 1 & 0 & 0 & 0 & -x_d^{(i)}x_s^{(i)} & -x_d^{(i)}y_s^{(i)} & -x_d^{(i)} \\ 0 & 0 & 0 & x_s^{(i)} & y_s^{(i)} & 1 & -y_d^{(i)}x_s^{(i)} & -y_d^{(i)}y_s^{(i)} & -y_d^{(i)} \\ \vdots & \vdots \\ x_s^{(n)} & y_s^{(n)} & 1 & 0 & 0 & 0 & -x_d^{(n)}x_s^{(n)} & -x_d^{(n)}y_s^{(n)} & -x_d^{(n)} \\ 0 & 0 & 0 & x_s^{(n)} & y_s^{(n)} & 1 & -y_d^{(n)}x_s^{(n)} & -y_d^{(n)}y_s^{(n)} & -y_d^{(n)} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

We use **Constrained Least Squares**, solve for  $Ah = 0$  such that  $\|h\|^2 = 1$ . Defining least squares problem:

$$\min_h \|Ah\|^2 \text{ such that } \|h\|^2 = 1$$

We know that:

$$\|Ah\|^2 = (Ah)^T(Ah) = h^T A^T Ah \text{ and } \|h\|^2 = h^T h = 1$$

We get the problem:

$$\min_h (h^T A^T Ah) \text{ such that } h^T h = 1$$

We define a **loss function**  $L(h, \lambda)$ :

$$L(h, \lambda) = h^T A^T Ah - \lambda(h^T h - 1)$$

Taking derivatives of  $L(h, \lambda)$  w.r.t  $h$ :

$$\begin{aligned} 2A^T Ah - 2\lambda h &= 0 \Rightarrow \\ A^T Ah &= \lambda h \end{aligned}$$

It is the Eigenvalue problem, where eigenvector  $h$  with smallest eigenvalue  $\lambda$  of matrix  $A^T A$  minimizes the loss function  $L(h, \lambda)$ .

Homography suffices to describe the transformation between two coordinate systems, for example, between two images, or between the world and an image projection of it in the following scenarios:

- Projecting 3D points on a **plane** to their image location.
- Projecting points in two different images that lie on the same plane.
- Projecting two images of a 3D object where the camera has been rotated but has not been translated.
- Projecting general 3D motion of images for small viewpoint changes.

## Dealing with Outliers – RANSAC

When we computed the homography matrix, we took all the pairs of matching features, but not all of these pairs necessarily correspond to valid matches. If the **number of outliers is lower than 50%** we can use **RANSAC**.

### RANDOM SAMPLE CONSENSUS (RANSAC)

General RANSAC Algorithm:

1. Randomly choose  $s$  samples. Typically  $s$  is the minimum samples to fit the model.
2. Fit the model to the randomly chosen samples.
3. Count the number  $M$  of data points (inliers) that fit the model (and measuring some distance) within a measure error  $\epsilon$ .
4. Repeat Steps 1-3  $N$  times.
5. Choose the model that has the largest number  $M$  of inliers.

For homography  $s = 4$  points (for affine transform  $s = 3$ , for a line  $s = 2$ ), and  $\epsilon$  is acceptable alignment error in pixels.

#### Parameters

- $N$  is the amount of data we have (we can say samples, but don't confuse with  $s$ ).
- $s$  is the number of points in the data we have.
- $e$  is the probability that point  $s$  is an outlier (hence,  $1 - e$  is the probability that  $s$  is inlayer).
- $p$  is the desired probability that we get a good sample.

We usually choose  $N$  such that with  $p = 0.99$  at least one inlayer in our data. We can compute the ideal  $N$  using the fact that we want  $(1 - e)^s$  (probability that we choose  $s$  inlayers), and for  $N$  samples such that we have at least one outlier is  $(1 - (1 - e)^s)^N$ . The probability that we are looking for is (that at least one sample of  $s$  points is composed of only inliers):

$$p = 1 - (1 - (1 - e)^s)^N$$

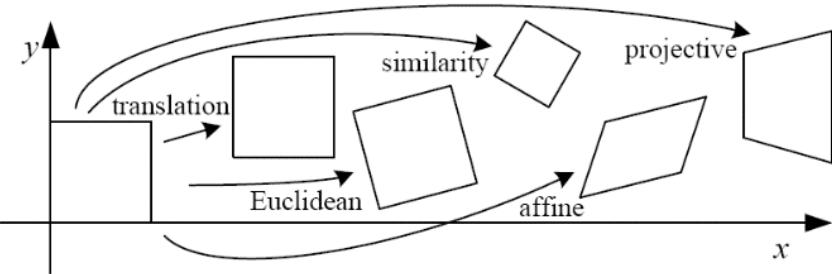
So:

$$\log(1 - p) = N \log(1 - (1 - e)^s) \Rightarrow N = \frac{\log(1 - p)}{\log(1 - (1 - e)^s)}$$

s	proportion of outliers $e$						
	5%	10%	20%	25%	30%	40%	50%
2	2	3	5	6	7	11	17
3	3	4	7	9	11	19	35
4	3	5	9	13	17	34	72
5	4	6	12	17	26	57	146
6	4	7	16	24	37	97	293
7	4	8	20	33	54	163	588
8	5	9	26	44	78	272	1177

$$p = 0.99$$

How big is  $s$ ?



Name	Matrix	# D.O.F.	Preserves:	Icon
translation	$\begin{bmatrix} \mathbf{I} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	2	orientation + ...	
rigid (Euclidean)	$\begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	3	lengths + ...	
similarity	$\begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix}_{2 \times 3}$	4	angles + ...	
affine	$\begin{bmatrix} \mathbf{A} \end{bmatrix}_{2 \times 3}$	6	parallelism + ...	
projective	$\begin{bmatrix} \tilde{\mathbf{H}} \end{bmatrix}_{3 \times 3}$	8	straight lines	

### Example

$N$  is 12 points, minimal sample size  $s = 2$  (we want to fit a line). The data is contaminated with 2 outliers, hence  $e = \frac{2}{12} > 20\%$ . So, for probability  $p = 0.99$ :

$$N = \frac{\log(1 - p)}{\log(1 - (1 - e)^s)} = \frac{\log(1 - 0.99)}{\log(1 - (1 - 0.2)^2)} \approx 5$$

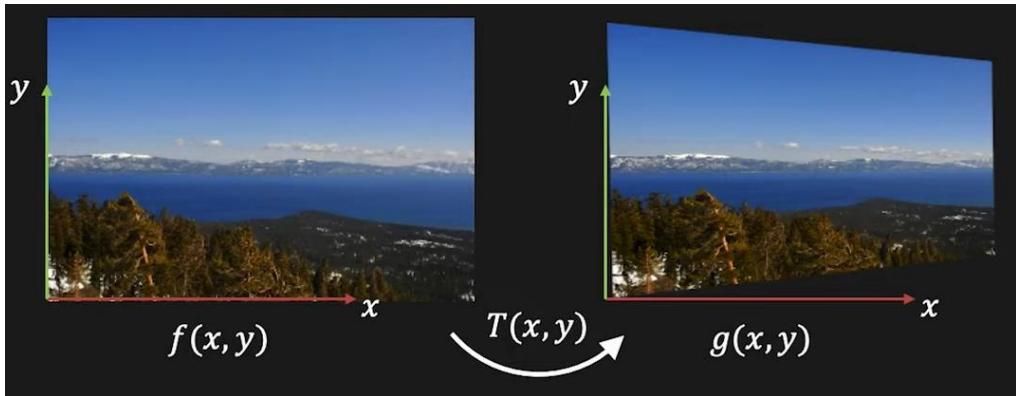
This is compared to  $N = 66$  if we have tried to use all the pairs of points.

## Warping and Blending Images

### WARPING IMAGES

Given a transformation  $T$  and image  $f(x, y)$ , compute the transformation  $g(x, y)$ :

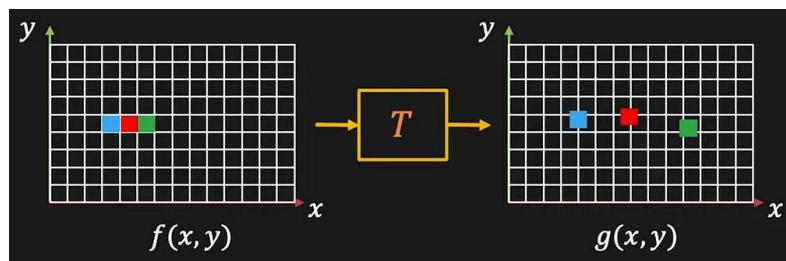
$$g(x, y) = f(T(x, y))$$



### Forward Warping

Send each pixel  $(x, y)$  in  $f(x, y)$  to its corresponding location  $T(x, y)$  in  $g(x, y)$ .

$$g(x, y) = f(T(x, y))$$

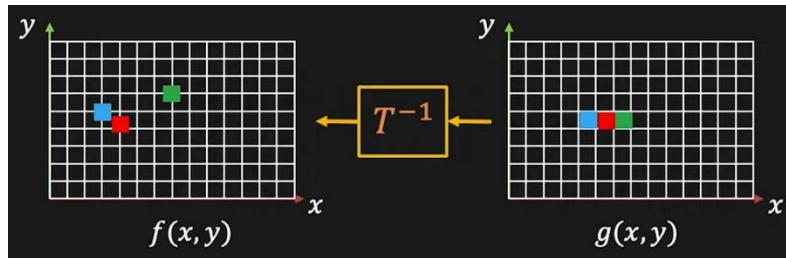


The pixels that we transform, may not land exactly where it is needed in  $g(x, y)$  (can land between pixels), also not all pixels in  $g(x, y)$  are filled, which **can result in holes**.

### Backward Warping

To solve this issue we use **Backward Warping**. Get each pixel  $(x, y)$  in  $g(x, y)$  from its corresponding location  $T^{-1}(x, y)$  in  $f(x, y)$ .

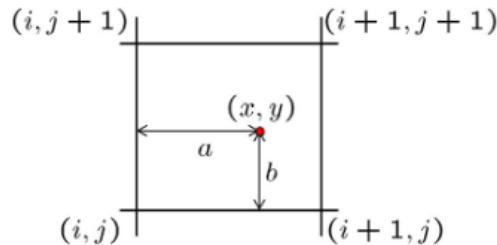
$$g(x, y) = f(T^{-1}(x, y))$$



First, we take our image  $f(x, y)$  and we apply the forward warp to the four corners of  $f(x, y)$ , which gives us four corners of  $g(x, y)$  (bounding box for  $g(x, y)$ ). We can use **interpolation** to pick the best value for the pixel we want in  $g(x, y)$  or use **Nearest Neighbor**.

### Bilinear Interpolation

Sampling at  $f(x, y)$  (pixels  $(i, j), (i + 1, j), (i, j + 1), (i + 1, j + 1)$  are known):



$$f(x, y) = [(1 - a)(1 - b)f(i, j)] + [a(1 - b)f(i + 1, j)] + [(1 - a)bf(i, j + 1)] + [abf(i + 1, j + 1)]$$

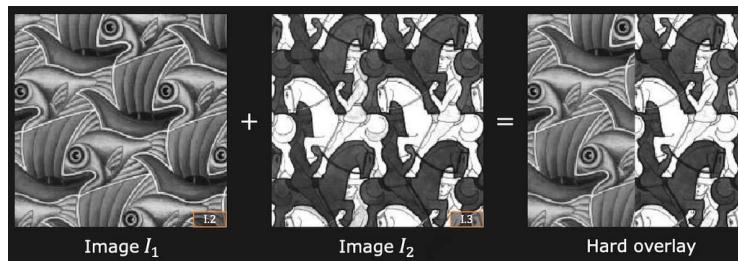
## BLENDING IMAGES

At the result of alignment we will see **Hard Seams**, due to vignetting, exposure differences, etc.



If we have a stack of images we can use the average of those images to try and remove the seams, but they are still visible (the human eye is extremely sensitive to brightness changes).

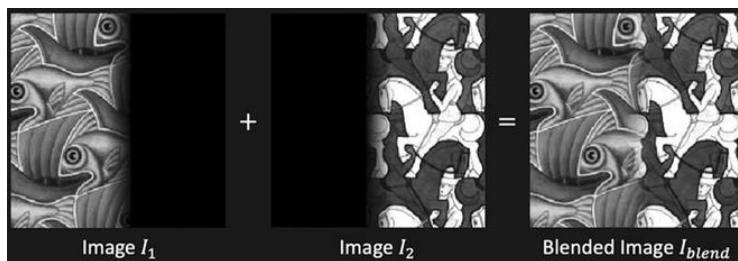
Say we want to blend images  $I_1$  and  $I_2$  at the center, we can just cut and paste the halves (**hard overlay**):



Using some mask  $M(x, y)$ , source image -  $S(x, y)$  and target image -  $T(x, y)$ :

$$I_{blend}(x, y) = \begin{cases} S(x, y), & M(x, y) = 1 \\ T(x, y), & M(x, y) = 0 \end{cases}$$

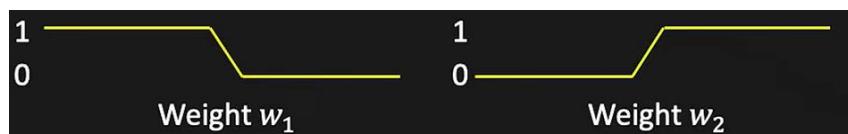
Or we could use a weight function, which gives a better result:



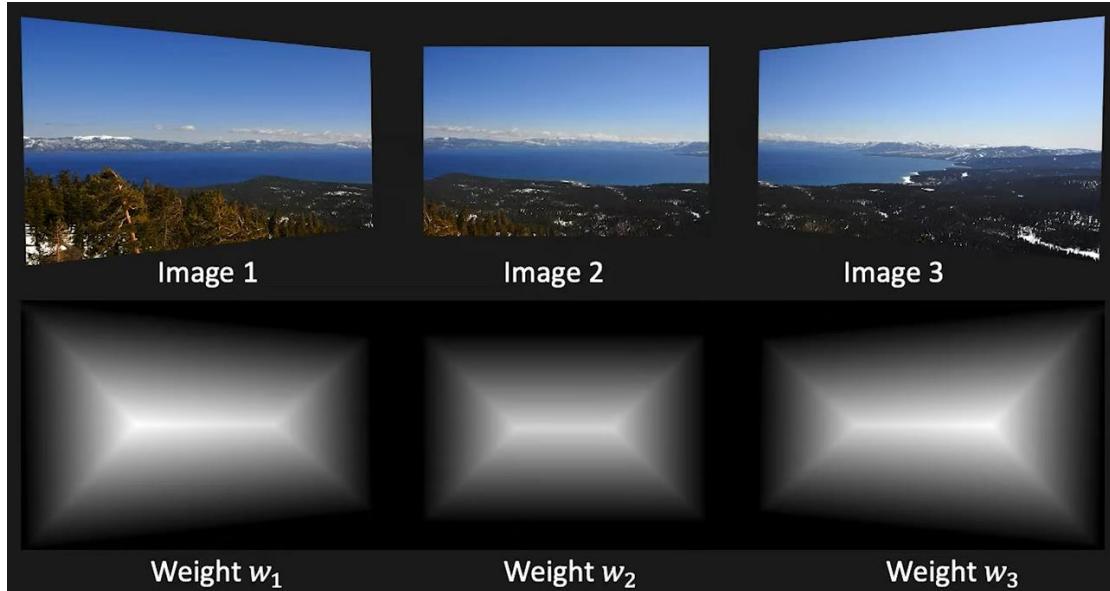
Such that:

$$I_{blend} = \frac{w_1 I_1 + w_2 I_2}{w_1 + w_2}$$

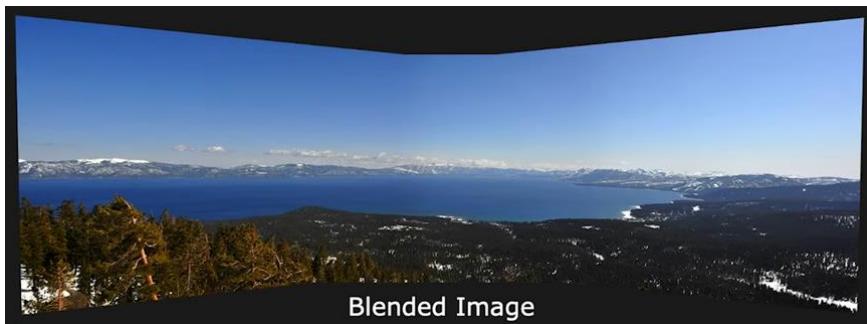
Where  $w_1$  and  $w_2$  are some gradual weights, like:



We can use blending of images. Where we first compute the weighting functions for all images, such that pixels closer to the edge get a lower weight (we can use **Distance Transform**).



The result look more realistic:



Another idea is using **Multi-Resolution Blending with a Laplacian Pyramid**.

*Multi-Resolution Blending with a Laplacian Pyramid*

The idea is to use **wide transition regions** for **low-frequency components** and **narrow transition regions** for **high-frequency components** (those are **edges**).

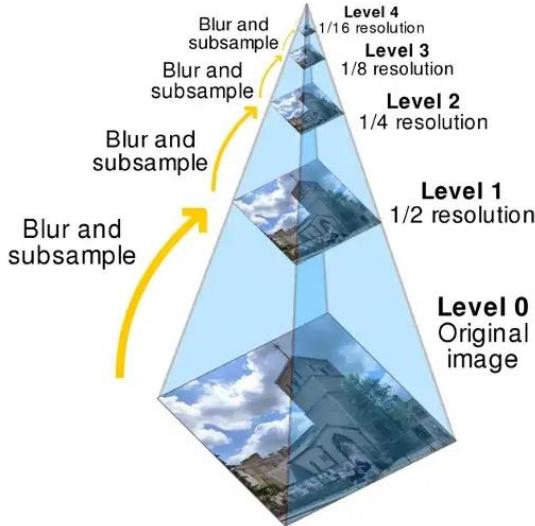
Gaussian Pyramid

Choose some Gaussian kernel, for example  $k$  is  $5 \times 5$  Gaussian filter. Then make a hierarchy of images:

$$G_0 = \text{Original Image (Full Resolution)}$$

$$G_i = (k * G_{i-1})_{\downarrow 2}$$

Where  $\downarrow 2$  is downsampling by factor of 2 in both dimensions.



After this step we want to find the edges that are important at each of these scales, we do it using the difference of Gaussians (DoG) at each scale, we get some sort of high-pass image:

$$L_i = G_i - (k * G_i)$$

Where  $\{L_i\}$  form a **Laplacian pyramid**.

We can recover the original image as:

$$I = \sum_{i=0}^N (L_i) \uparrow \text{To full size}$$

i.e. add back all the edges at different scales, where the base image is the smallest (blurriest image)  $G_N = L_N$ .

To do the image composition we compute the Laplacian pyramids for  $S(x, y)$  and  $T(x, y)$ , those are:  $L^S, L^T$ . And we compute Gaussian pyramid for the mask  $M(x, y)$  this is  $G(x, y)$ . And the Laplacian pyramid for composite:

$$L_i^I = G_i L_i^S + (1 - G_i) L_i^T, \quad i = 0, 1, \dots, N$$

And we add up to get final composite.

Algorithm:

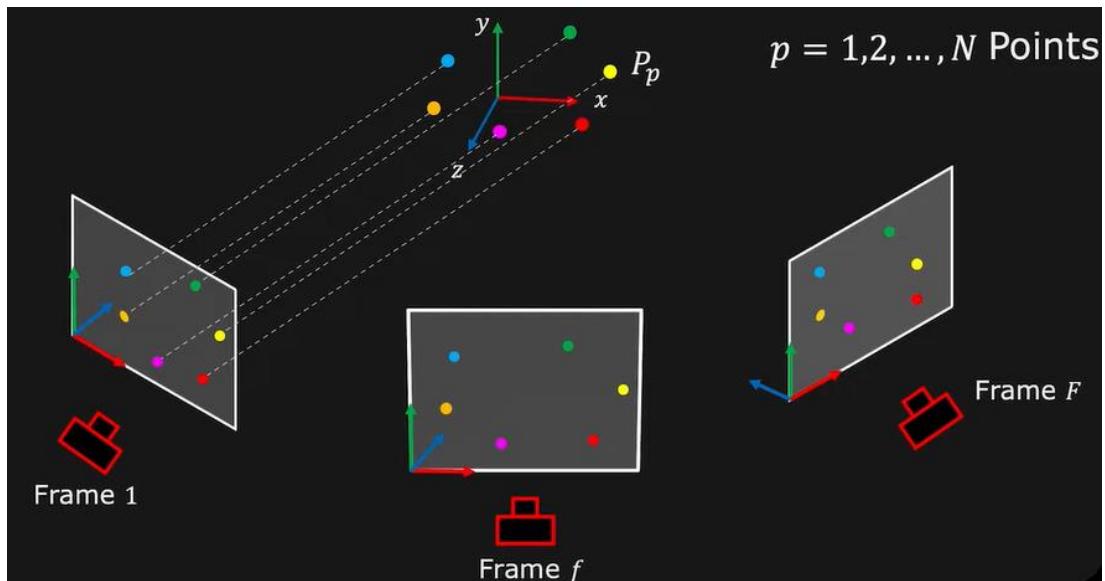
1. Build a Gaussian pyramid for each image.
2. Build the Laplacian pyramid for each image.
3. Decide/find the blending border (in the example: left half belongs to image 1, and right half to image 2, such that the blending border is cols/2)
  - o Split by index, or
  - o Split using a 2 masks (can be weighted masks)
4. Construct a new mixed pyramid - mix each level separately according to 3.
5. Reconstruct a blend image from the mixed pyramid.

## STRUCTURE FROM MOTION (SfM)

**Structure from Motion** (SfM) is a technique used to estimate the 3D structure of a scene from a series of 2D images. It involves analyzing the motion of an object or camera across multiple frames to infer the 3D structure of the scene being captured. This is achieved by identifying common features across multiple images and using the motion of these features to triangulate the position of the features in 3D space. SfM can be used for a variety of applications, including 3D reconstruction, object recognition, and autonomous navigation.

Given sets of corresponding Image points (2D):  $(u_{f,p}, v_{f,p})$ .

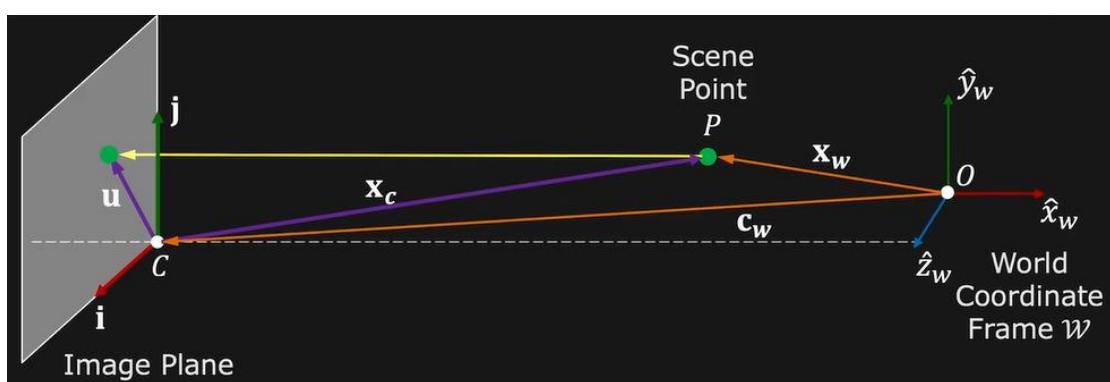
Find scene points (3D):  $P_p$  assuming **orthographic camera**.



### From 3D to 2D: Orthographic Projection

Perspective cameras exhibit orthographic projection when distance of scene from camera is large compared to depth variation within scene (magnification is nearly constant).

$$\begin{cases} u = i \cdot x_c = i^T x_c \\ v = j \cdot x_c = j^T x_c \end{cases}$$



Expressing the point in world coordinate frame:

$$\begin{aligned} u &= i^T x_c = i^T(x_w - c_w) = i^T(P - C) \\ v &= j^T x_c = j^T(x_w - c_w) = j^T(P - C) \end{aligned}$$

So, in orthographic SfM we are:

- Given sets of corresponding Image points (2D):  $(u_{f,p}, v_{f,p})$
- Find **scene points** (3D):  $\{P_p\}$
- Camera **positions**:  $\{C_f\}$ , camera **orientations**  $\{(i_f, j_f)\}$  are unknown

We can write the image of point  $P_p$  in camera frame  $f$  as:

$$\begin{aligned} u_{f,p} &= i_f^T (P_p - C_f) \\ v_{f,p} &= j_f^T (P_p - C_f) \end{aligned}$$

Where only  $u_{f,p}$  and  $v_{f,p}$  are known to us. But we can remove  $C_f$  from equations to simplify the SfM problem. We will use the **Centering Trick**:

The points in scene are projected orthographically to frame  $f$ , and we assume that the origin of world at centroid of scene points:

$$\frac{1}{N} \sum_{p=1}^N P_p = \bar{P} = 0$$

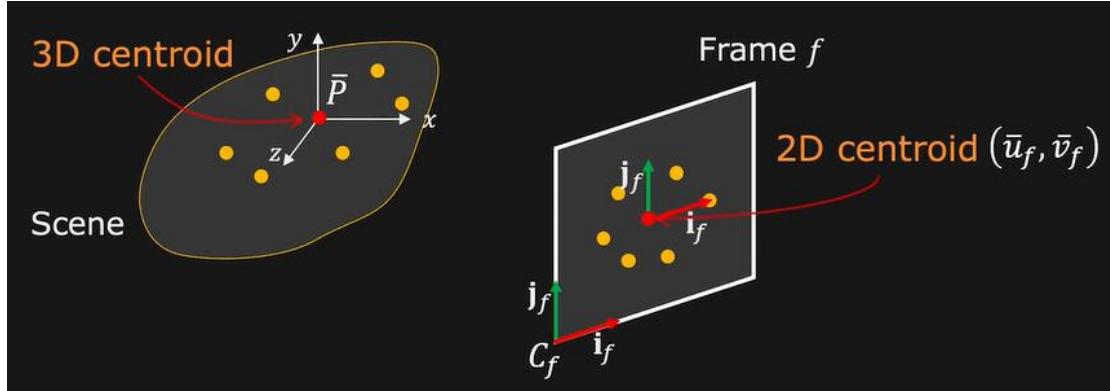
We will compute the scene points w.r.t their centroid! Centroid  $(\bar{u}_f, \bar{v}_f)$  of the image points in frame  $f$ :

$$\begin{aligned} \bar{u}_f &= \frac{1}{N} \sum_{p=1}^N u_{f,p} = \frac{1}{N} \sum_{p=1}^N i_f^T (P_p - C_f) = \frac{1}{N} i_f^T \sum_{p=1}^N P_p - \frac{1}{N} i_f^T C_f \sum_{p=1}^N 1 \xrightarrow{\frac{1}{N} \sum_{p=1}^N P_p = 0} \\ &\Rightarrow -\frac{1}{N} i_f^T C_f \sum_{p=1}^N 1 = -i_f^T C_f \\ \bar{v}_f &= \frac{1}{N} \sum_{p=1}^N v_{f,p} = \frac{1}{N} \sum_{p=1}^N j_f^T (P_p - C_f) = \frac{1}{N} j_f^T \sum_{p=1}^N P_p - \frac{1}{N} j_f^T C_f \sum_{p=1}^N 1 \xrightarrow{\frac{1}{N} \sum_{p=1}^N P_p = 0} \\ &\Rightarrow -\frac{1}{N} j_f^T C_f \sum_{p=1}^N 1 = -j_f^T C_f \end{aligned}$$

So, it will be helpful to shift camera origin to the centroid  $(\bar{u}_f, \bar{v}_f)$ , the image points w.r.t.  $(\bar{u}_f, \bar{v}_f)$ :

$$\begin{aligned} \tilde{u}_{f,p} &= u_{f,p} - \bar{u} = i_f^T (P_p - C_f) - (-i_f^T C_f) = i_f^T P_p \\ \tilde{v}_{f,p} &= v_{f,p} - \bar{v} = j_f^T (P_p - C_f) - (-j_f^T C_f) = j_f^T P_p \end{aligned}$$

Camera location  $C_f$  now removed from equations.



## OBSERVATION MATRIX $W$

We can write:

$$\begin{bmatrix} \tilde{u}_{f,p} \\ \tilde{v}_{f,p} \end{bmatrix} = \begin{bmatrix} i_f^T \\ j_f^T \end{bmatrix} P_p$$

For  $N$  points and  $F$  frames:

$$\underbrace{\begin{bmatrix} \tilde{u}_{1,1} & \tilde{u}_{1,2} & \dots & \tilde{u}_{1,N} \\ \tilde{u}_{2,1} & \tilde{u}_{2,2} & \dots & \tilde{u}_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{u}_{F,1} & \tilde{u}_{F,2} & \dots & \tilde{u}_{F,N} \\ \tilde{v}_{1,1} & \tilde{v}_{1,2} & \dots & \tilde{v}_{1,N} \\ \tilde{v}_{2,1} & \tilde{v}_{2,2} & \dots & \tilde{v}_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{v}_{F,1} & \tilde{v}_{F,2} & \dots & \tilde{v}_{F,N} \end{bmatrix}}_{W_{2F \times N}} = \underbrace{\begin{bmatrix} i_1^T \\ i_2^T \\ \vdots \\ i_F^T \\ j_1^T \\ j_2^T \\ \vdots \\ j_F^T \end{bmatrix}}_{M_{2F \times 3}} \underbrace{\begin{bmatrix} P_1 & P_2 & \dots & P_N \end{bmatrix}}_{S_{3 \times N}}$$

Where each row (of  $F$  rows) is an image (frame) and each column (of  $N$  columns) is a point. The  $W_{2F \times N}$  is **Centroid-Subtracted Feature Points** (known),  $M_{2F \times 3}$  is a **Camera Motion** matrix (unknown) and  $S_{3 \times N}$  is **Scene Structure** (unknown).

SfM algorithm is based on the insight that the observation matrix has a special form, in particular it has a low rank.

### Rank of Observation Matrix

**Column Rank:** The number of linearly independent columns of the matrix.

**Row Rank:** The number of linearly independent rows of the matrix.

$$\begin{aligned} \text{Column Rank } (A) &= \text{Row Rank } (A) = \text{Rank}(A) \\ \text{Rank}(A) &\leq \min(m, n) \end{aligned}$$

Rank is the dimensionality of the space spanned by column or row vectors of the matrix.

**Properties:**

- $\text{Rank}(A^T) = \text{Rank}(A)$
- $\text{Rank}(A_{m \times n} B_{n \times p}) = \min(\text{Rank}(A_{m \times n}), \text{Rank}(B_{n \times p})) \leq \min(m, n, p)$
- $\text{Rank}(AA^T) = \text{Rank}(A^TA) = \text{Rank}(A^T) = \text{Rank}(A)$
- $A_{m \times m}$  is invertible iff  $\text{Rank}(A_{m \times m}) = m$

We have showed that:

$$W_{2F \times N} = M_{2F \times 3} \times S_{3 \times N}$$

And we know that:

$$\begin{aligned} \text{Rank}(MS) &\leq \text{Rank}(M), \quad \text{Rank}(MS) \leq \text{Rank}(S) \Rightarrow \\ \Rightarrow \text{Rank}(MS) &\leq \min(2F, 3), \quad \text{Rank}(MS) \leq \min(3, N) \Rightarrow \\ \Rightarrow \text{Rank}(W) &= \text{Rank}(MS) \leq \min(3, N, 2F) \end{aligned}$$

So:

$$\text{Rank}(W) \leq 3$$

## Tomasi-Kanade Factorization

Using **Singular Value Decomposition (SVD)** we know that for any matrix  $A$  there exist a factorization:

$$A_{M \times N} = U_{M \times M} \cdot \Sigma_{M \times N} \cdot V_{N \times N}^T$$

Where  $U$  and  $V^T$  are **orthonormal** and  $\Sigma$  is **diagonal**. Such that:

$$\Sigma_{M \times N} = \begin{pmatrix} \sigma_1 & 0 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & 0 & \dots & 0 \\ 0 & 0 & 0 & \sigma_4 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \sigma_N \\ 0 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots \end{pmatrix}$$

And  $\{\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_N\}$  are **Singular Values**. If  $\text{Rank}(A) = r$  the  $A$  has  $r$  **non-zero singular values**.

Using SVD:

$$W = U \Sigma V^T = \underbrace{(U)}_{2F \times 2F} \underbrace{\begin{pmatrix} \sigma_1 & 0 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & 0 & \dots & 0 \\ 0 & 0 & 0 & \sigma_4 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \sigma_N \\ 0 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots \end{pmatrix}}_{2F \times N} \underbrace{(V^T)}_{N \times N}$$

Where  $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \dots \geq \sigma_N$  are the **singular values** of  $W$ ,

and since  $\text{Rank}(W) \leq 3 \Rightarrow \text{Rank}(\Sigma) \leq 3$ . All except first 3 diagonal elements of  $\Sigma$  must be 0.

$$W = U\Sigma V^T = \underbrace{(U)}_{2F \times 2F} \underbrace{\begin{pmatrix} \sigma_1 & 0 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots \end{pmatrix}}_{2F \times N} \underbrace{(V^T)}_{N \times N}$$

So we can write as:

$$W = U_1 \Sigma_1 V_1^T = \underbrace{(U_1)}_{2F \times 3} \underbrace{\begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{pmatrix}}_{3 \times 3} \underbrace{(V_1^T)}_{3 \times N}$$

Using **Factorization**:

$$W = U_1 \Sigma_1^{\frac{1}{2}} \Sigma_1^{\frac{1}{2}} V_1^T = \left( U_1 \Sigma_1^{\frac{1}{2}} \right) \left( \Sigma_1^{\frac{1}{2}} V_1^T \right)$$

But, the decomposition is not unique, so for any  $3 \times 3$  non-singular matrix  $Q$ :

$$W = U_1 \Sigma_1^{\frac{1}{2}} Q Q^{-1} \Sigma_1^{\frac{1}{2}} V_1^T = \left( U_1 \Sigma_1^{\frac{1}{2}} Q \right) \left( Q^{-1} \Sigma_1^{\frac{1}{2}} V_1^T \right)$$

So, for some  $Q$ , we get:

$$M = U_1 \Sigma_1^{\frac{1}{2}} Q, \quad S = Q^{-1} \Sigma_1^{\frac{1}{2}} V_1^T$$

Using the orthonormality of the motion matrix  $M$ :

$$M = \begin{bmatrix} i_1^T \\ i_2^T \\ \vdots \\ i_F^T \\ j_1^T \\ j_2^T \\ \vdots \\ j_F^T \end{bmatrix} = \underbrace{U_1 \Sigma_1^{\frac{1}{2}}}_{\text{Computed}} Q = \begin{bmatrix} \hat{i}_1^T \\ \hat{i}_2^T \\ \vdots \\ \hat{i}_F^T \\ \hat{j}_1^T \\ \hat{j}_2^T \\ \vdots \\ \hat{j}_F^T \end{bmatrix} \quad Q = \begin{bmatrix} i_1^T Q \\ i_2^T Q \\ \vdots \\ i_F^T Q \\ j_1^T Q \\ j_2^T Q \\ \vdots \\ j_F^T Q \end{bmatrix}$$

With **orthonormality constraints**:

$$\begin{cases} i_f \cdot i_f = i_f^T i_f = 1 \\ j_f \cdot j_f = j_f^T j_f = 1 \\ i_f \cdot j_f = i_f^T j_f = 0 \end{cases} \Rightarrow \begin{cases} \hat{i}_f^T Q Q^T \hat{i}_f = 1 \\ \hat{j}_f^T Q Q^T \hat{j}_f = 1 \\ \hat{i}_f^T Q Q^T \hat{j}_f = 0 \end{cases}$$

Where we have computed  $(\hat{i}_f^T, \hat{j}_f^T)$  for  $f = 1, 2, \dots, F$  and  $Q$  is unknown. But  $Q$  is  $3 \times 3$  matrix, 9 variables,  $3F$  quadratic equations.  $Q$  can be solved with 3 or more images ( $F \geq 3$ ) using **Newton's method**.

Summarizing the **Orthographic SfM**:

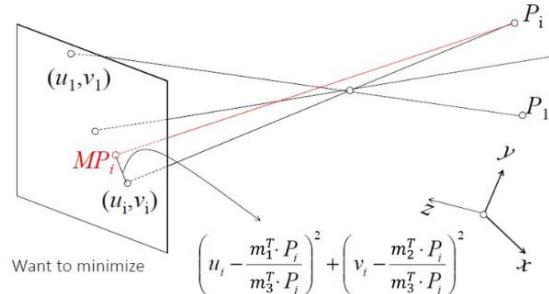
1. Detect and track feature points.
2. Create the centroid subtracted matrix  $W$  of corresponding feature points.
3. Compute SVD of  $W$  and enforce rank constraint.

$$W = U\Sigma V^T = \underbrace{U_1}_{2F \times 3} \underbrace{\Sigma_1}_{3 \times 3} \underbrace{V_1^T}_{3 \times P}$$

4. Set  $M = U_1 \Sigma_1^{\frac{1}{2}} Q$  and  $S = Q^{-1} \Sigma_1^{\frac{1}{2}} V_1^T$ .
5. Find  $Q$  by enforcing the orthonormality constraint.

## Bundle Adjustment

Here we assume we have  $N$  cameras observing a stationary 3D scene of  $n$  points. But not all of  $P$  points (landmarks) are observed in each image. This can happen due to occlusions or noise (e.g. missed detections). The objective function is given by **minimizing the re-projection error** (distance between observed feature and projected 3D point in image plane) w.r.t camera parameters and 3D point cloud for all observed projections of the  $n$  points.



Let  $\Pi = \{\pi_i\}$  denote  $N$  cameras including their intrinsic and extrinsic parameters, let  $P_w = \{x_p^w\}$  with  $x_p^w \in \mathbb{R}^3$  denote the set of  $P$  3D points in world coordinates, let  $\chi_s = \{x_{ip}^s\}$  with  $x_{ip}^s \in \mathbb{R}^2$  denote the image (screen) observations in all  $i$  cameras.

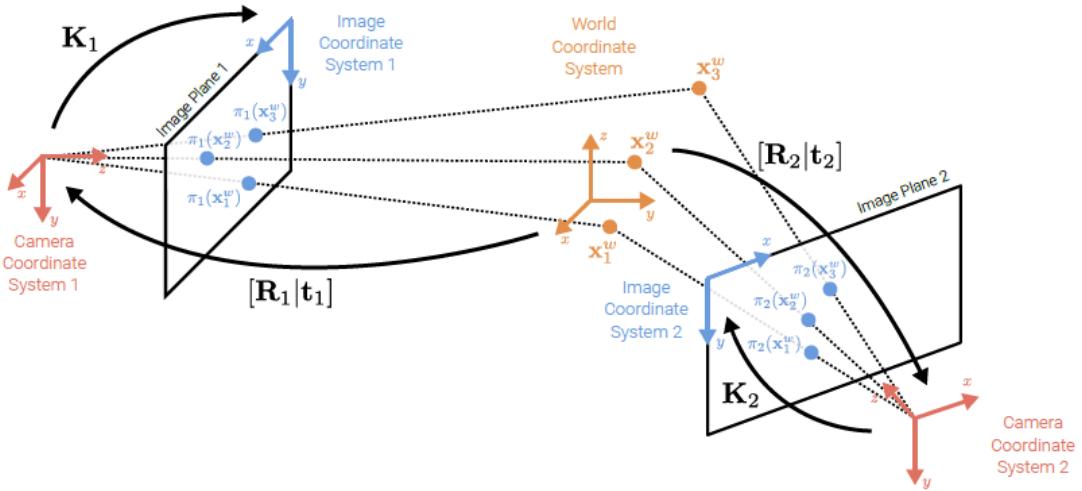
**Bundle adjustment** minimizes the reprojection error of all observations:

$$\Pi^*, P_w^* = \arg \min_{\Pi, P_w} \sum_{i=1}^N \sum_{p=1}^P w_{ip} \|x_{ip}^s - \pi_i(x_p^w)\|_2^2$$

Here,  $w_{ip}$  indicates if point  $p$  is observed in image  $i$  and  $\pi_i(x_p^w)$  is the 3D-to-2D projection of 3D world point  $x_p^w$  onto the 2D image plane of the  $i^{th}$  camera, i.e.:

$$\pi_i(x_p^w) = \begin{pmatrix} \tilde{x}_p^s \\ \tilde{w}_p^s \\ \tilde{y}_p^s \\ \tilde{w}_p^s \end{pmatrix}, \quad \tilde{x}_p^s = K_i(R_i x_p^w + t_i)$$

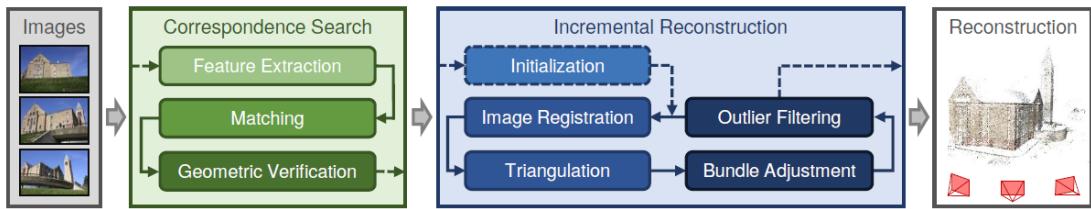
$K_i$  and  $[R_i|t_i]$  are the intrinsic and extrinsic parameters of  $\pi_i$ , respectively. During bundle adjustment, we optimize  $\{(K_i, R_i, t_i)\}$  and  $\{x_p^w\}$  jointly. An illustration of the bundle adjustment problem involving three points and two cameras can be seen in the following figure:



As the energy landscape of the **bundle adjustment problem** is **highly non-convex**, a good **initialization is crucial** to avoid getting trapped in bad local minima. Initializing all 3D points and cameras jointly is difficult (occlusion, viewpoint, matching outliers), so **incremental bundle adjustment** is used, which means initializing with a carefully selected two-view reconstruction and iteratively adding new images/cameras. After initialization, the actual **optimization is also challenging**. Given millions of features and thousands of cameras, large-scale bundle adjustment is **computationally demanding** (**cubic complexity** in the number of unknowns). Luckily, the **problem is sparse** (not all 3D points are observed in every camera), and efficient sparse implementations (e.g., Ceres) can be exploited in practice.

## INCREMENTAL STRUCTURE-FROM-MOTION

In the following figure the SfM pipeline COLMAP is shown:



The pipeline is divided into two fundamental parts: **Correspondence Search** and **Incremental Reconstruction**.

**Correspondence Search** is the stage where 2D features are **found and matched** in and between the images.

**Incremental Reconstruction** is the stage where these matched features are used to **infer 3D structure and camera parameters**. It is **incremental** in the sense that we **start with only two cameras and then add new cameras incrementally**.

The **correspondence search stage** begins with **feature extraction** in all images. For this, SIFT or another feature detector such as SURF or BRISK can be used. The **next step (matching)** is to find overlapping **image pairs and associated feature correspondences**. The first step of the **incremental reconstruction stage** (initialization) begins with two images (with matched features). These features are processed to **estimate the epipolar geometry**. When this is done, the **next step (image registration)** starts for the first time and a new image is selected that has correspondences in the current set of images. Then, the **camera pose for this new image is estimated**. Let  $\chi = \{\bar{x}_i^S, \bar{x}_i^W\}_{i=1}^N$  be a set of  $N$  3D-to-2D correspondences related by  $\bar{x}_i^S = P\bar{x}_i^W$ . As the correspondence vectors are homogeneous, they have the same direction but differ in magnitude. Thus, the equation above can be expressed as  $\bar{x}_i^S \times P\bar{x}_i^W = 0$ . Using the **Direct Linear Transform**, this can be written as a **linear equation in the entries of  $P$** . The **solution** to the constrained system (fixing  $P$ 's scale) is **given by SVD**. Given that  $P = K[R|t]$  and  $K$  is upper-triangular, both  $K$  and  $R$  can be easily obtained from the front  $3 \times 3$  submatrix of  $P$  using standard RQ factorization. If  $K$  is known, we can even estimate  $P$  from only three points (P3P algorithm). In practice, random sampling consensus (RANSAC) is additionally used to remove outliers. The **next step** of the pipeline is **triangulation**. Given the newly registered image, new correspondences can be triangulated. In COLMAP, a robust triangulation method is proposed that also handles outliers. After this, the bundle adjustment step is performed. Since incremental SfM only affects the model locally, COLMAP performs local BA on the locally connected images, and global BA only once in a while for efficiency. For solving the sparse large-scale optimization problem, COLMAP uses **Ceres**. After each BA, observations with large reprojection errors and cameras with abnormal field of views or large distortion coefficients are removed. COLMAP also features an additional multi-view stereo stage to obtain dense geometry, a result is shown in the following figure.



## SfM – Example Exercise

1. A camera is rotating around its axis in a fixed angular velocity and taking photos of a palace from within. Is it possible to reconstruct the depth of the objects in the scene based on the taken pictures? Write down the mathematical relationship between the points in the image taken at time  $t$  and the points in the image taken at  $t + \Delta t$ , when the camera has moved by an angle of  $\theta$ .
2. Now the camera is mounted on a drone and the drone is moving around the palace in a circular orbit (the drone's position is unknown). Is it possible to reconstruct the 3D shape of the palace?
3.  $N$  camera matrices  $M_1, \dots, M_N \in \mathbb{R}^{3 \times 4}$  are given. Write down the equations system for triangulating a 3D point  $P$  by using 2D point-correspondences  $(p_1, p_2, \dots, p_N)$  from  $N$  different images and describe how you can solve it.
4. Can you use the drone-mounted camera for reconstructing the 3D shape of a bird flapping her wings, at each point of time while it's flying?

### Solution:

1. We **can't recover the depth of the objects**. When the two views are separated by a pure rotation, the mapping of the points in the images are given by a homography.

$$H = KR_\theta^{-1}K^{-1}$$

2. Now we can recover the depth objects in the scene (specifically - the shape of the palace). This is the classic SfM setup we've just learned, where we can infer both the camera matrices and the 3D shape of the objects based on keypoint matching.
3. As denoted before:

$$\begin{aligned} p_i &= (x_i, y_i)^T \\ M_i &= \begin{bmatrix} - & m_{i,1}^T & - \\ - & m_{i,2}^T & - \\ - & m_{i,3}^T & - \end{bmatrix} \end{aligned}$$

We can now write the triangulation equation with  $N$  cameras for a single point  $P$  by extending the equation from the lecture and this tutorial for more matches:

$$\begin{bmatrix} y_1 m_{1,3}^T - m_{1,2}^T \\ m_{1,1}^T - x_1 m_{1,3}^T \\ \vdots \\ y_N m_{N,3}^T - m_{N,2}^T \\ m_{N,1}^T - x_N m_{N,3}^T \end{bmatrix} P = AP = 0$$

The solution is given by SVD (homogeneous linear system).

4. Since we don't have a rigid body assumption on the bird's motion, we will not be able to reconstruct the shape of the bird using the proposed method. Our basic assumption in SfM is that the **scene is static**, and when the bird is flapping her wings the motion of the points will not be described by a simple matrix.

## Object Tracking

We want to track the location of target objects in each frame of a video sequence.

### OBJECT TRACKING VS. OBJECT DETECTION

Why can't we use object detection in each frame in the whole video and track the object?

- If the image has **multiple objects**, then we have no way of connecting the objects in the current frame to the previous frames.
- If the object you were tracking goes **out of the camera view** for a few frames and another one appears, we have no way of knowing if it's the same object.
- Essentially, during **detection**, we work with one image at a time and we have **no idea about the motion and past movement of the object**, so we can't uniquely track objects in a video.

Whenever there is a moving object in the videos, there are certain cases when the visual appearance of the object is not clear. In all such cases, detection would fail while tracking succeeds as it also has the motion model and history of the object.

### CHANGE DETECTION

Given a static cameras observing a scene (room, street, etc.), we want to find meaningful changes (moving objects, people, etc.). The real problem here is the robust and real-time classification of each pixel as "**foreground**" (motion/change) or "**background**" (static).

We want to ignore uninteresting changes, like:

- Background fluctuations (waves in the sea, trees moving in a wind, etc.)
- Image noise (at night the light levels are low, so the noise levels are relatively high)
- Rain, snow, turbulence (mirage effect)
- Illumination changes & shadows
- Camera shake

### *Detection Failure Cases*

- **Occlusion** – the object is partially or completely occluded by some other object.
- **Identity Switches** – usually happens after two objects cross each other, can't tell the correct identity.
- **Motion Blur** – object is blurred due to the motion of the object or camera. Hence, visually, the object doesn't look the same anymore.
- **Viewpoint Variation** – different viewpoint of an object may look very different visually and without the context it becomes very difficult to identify the object using only visual detection.
- **Scale Change** – huge changes in object scale may cause a failure in detection.
- **Background Clutters** – background near object has similar color or texture as the target object. Hence, it may become harder to separate the object from the background.

- **Illumination Variation** – illumination near the target object is significantly changed.  
Hence, it may become harder to visually identify it.
- **Low Resolution** – when the number of pixels inside the ground truth bounding box is low, it may be too hard to detect the objects visually.
- A good object tracker has two basic models: **Motion Model** and **Visual Appearance Model**.

## MOTION MODEL

It is the **ability to understand and model the motion of the object**. A good motion model captures the dynamic behavior of an object. It predicts the potential position of objects in the future frames, hence, reducing the search space. However, the motion model alone can fail in scenarios where motion is caused by things that are not in a video or abrupt direction and speed change. Some of the classic methods understand the motion pattern of the object and try to predict that. However, the problem with such approaches is that they can't predict the abrupt motion and direction changes. Examples of such techniques are:

- Optical Flow
- Kalman Filtering
- Kanade-Lucas-Tomashi (KLT) feature tracker
- Mean shift tracking

## VISUAL APPEARANCE MODEL

It is the ability to understand the appearance of the object that is being tracked. Trackers need to learn to discriminate the object from the background. In **single object trackers** (one object), visual appearance alone could be enough to track the object across frames, while in **multiple-object trackers**, visual appearance alone is not enough.

## SIMPLE FRAME DIFFERENCE

The simplest solution is to label significant difference between current and previous frames as foreground.

$$F_t = |I_t - I_{t-1}| > T, \quad T: \text{threshold}$$

This technique isn't robust to the challenges we have stated above, and also for example for a large car of same color, the pixels (that aren't the edges of the car) will be labeled as background.

A more robust way is to build a simple model of background before classification. Where:

- Background  $B$  is the **average** of first  $K$  frames:  $\{I_1, I_2, \dots, I_K\}$
- Input frame  $I_t$
- Foreground  $F_t$  is computed as:  $F_t = |I_t - B| > T, \quad T: \text{threshold}$

But this technique cannot handle change in lighting, background, etc.

We can change this technique to use the median. Where:

- Background  $B$  is the **median** of first  $K$  frames:  $\{I_1, I_2, \dots, I_K\}$

- Input frame  $I_t$
- Foreground  $F_t$  is computed as:  $F_t = |I_t - B| > T, T: threshold$

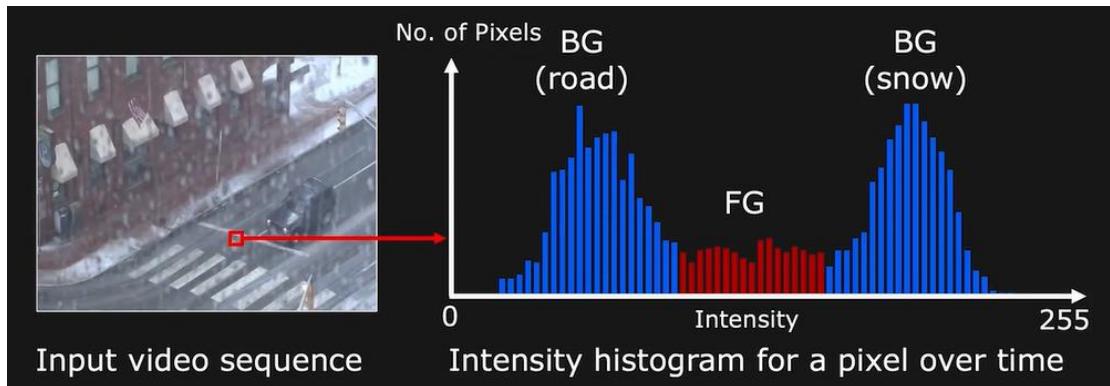
So the model should adapt as the scene changes with time. We can build a simple **adaptive model** of background over time. Where:

- Background  $B_t$  is the **median** of last  $K$  frames:  $\{I_{t-1}, I_{t-2}, \dots, I_{t-K}\}$
- Input frame  $I_t$
- Foreground  $F_t$  is computed as:  $F_t = |I_t - B_t| > T, T: threshold$

But this model cannot handle significant pixel fluctuations (weather, shadows, shake, etc.).

## MIXTURE MODEL

Intensity distribution at each pixel over time.



The intensity variations due to static scene (**road**), noise (**snow**), and occasional moving objects (**vehicles**). Our intuition states that pixels are background most of the time.

Using the **Gaussian Model**:

**1D Gaussian:**

$$\omega \cdot \eta(x, \mu, \sigma) = \omega \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$$

We assume that  $P(x)$  is made of  $K$  different Gaussians.

**Gaussian Mixture Model (GMM) Distribution** – weighted sum of  $K$  Gaussians:

$$P(x) = \sum_{k=1}^K \omega_k \eta_k(x, \mu_k, \sigma_k)$$

Such that:

$$\sum_{k=1}^K \omega_k = 1$$

*Higher Dimensional GMM*

Let  $P(x)$  be the probability distribution of a  $D$ -dimensional random variable  $x \in \mathbb{R}^D$ . For example:  $x = [r, g, b]^T$

GMM of  $P(x)$  is the sum of  $K$   $D$ -dimensional Gaussians:

$$P(x) = \sum_{k=1}^K \omega_k \eta_k(x, \mu_k, \sigma_k)$$

Such that:

$$\sum_{k=1}^K \omega_k = 1$$

Where:

$$\eta(x, \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{D}{2}} |\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu)^T (\Sigma)^{-1} (x-\mu)}$$

Mean  $\mu = \begin{bmatrix} \mu_r \\ \mu_g \\ \mu_b \end{bmatrix}$ , Covariance matrix  $\Sigma = \begin{pmatrix} \sigma_r^2 & 0 & 0 \\ 0 & \sigma_g^2 & 0 \\ 0 & 0 & \sigma_b^2 \end{pmatrix}$  (can be a full matrix). This GMM can be

estimated from  $P(x)$ .

Given a GMM for intensity/color variation at a pixel over time, we want to classify individual Gaussians as foreground/background. Using our intuition, that states that pixels are background most of the time. That is, Gaussians with large supporting evidence  $\omega$  and small  $\sigma$ .

$$\text{Large } \frac{\omega}{\sigma} : \text{Background}, \quad \text{Small } \frac{\omega}{\sigma} : \text{Foreground}$$

The algorithm:

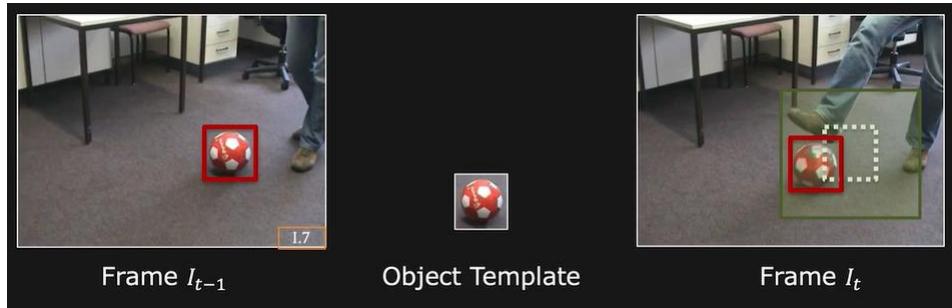
1. Compute pixel color histogram  $H$  using first  $N$  frames.
2. Normalize histogram:  $\hat{H} \leftarrow \frac{H}{\|H\|}$
3. Model  $\hat{H}$  as mixture of  $K$  (3 to 5) Gaussians.
4. For each subsequent frame:
  - a. The pixel value  $x$  belongs to Gaussian  $k$  in GMM for which  $\|x - \mu_k\|$  is minimum and  $\|x - \mu_k\| < 2.5\sigma_k$ .
  - b. If  $\frac{\omega_k}{\sigma_k}$  is large, then classify pixel as background. Else classify as foreground.
  - c. Update histogram  $H$  using new pixel intensity.
  - d. If  $\hat{H}$  and  $\frac{H}{\|H\|}$  differ a lot ( $\|\hat{H} - \frac{H}{\|H\|}\|$  is large),  $\hat{H} \leftarrow \frac{H}{\|H\|}$  and refit GMM.

## OBJECT TRACKING USING TEMPLATE MATCHING

We can use the meaningful changes that we found to define an object of interest (or ROI) and to track the object through the entire video sequence. Given a location of target in initial or previous frame, we want to find the location of target in current frame.

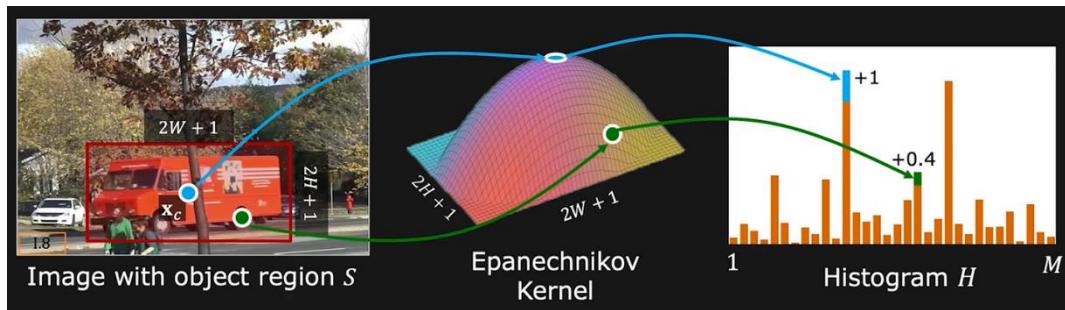
We can do template matching using the image region with the appearance of the object to create a templates and apply template matching.

Given template window  $S$  in frame  $I_{t-1}$ , search the neighborhood to find match in image  $I$ .



It's a simple implementation, but not robust to change in scale, viewpoint, occlusions etc.

Another way is using a **histogram-based tracking**. We know that the more reliable points within the region of interest tend to be closer to the center of the region. We can use **Weighted Histogram**, that gives more importance to pixels at the center.



For example, using **Epanechnikov kernel**:

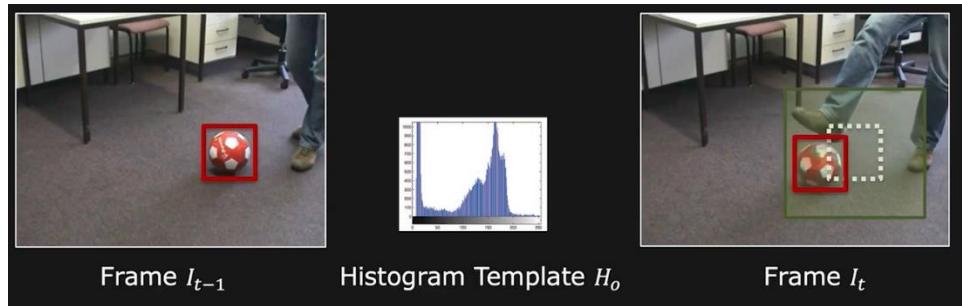
$$k(\tilde{x}) = \begin{cases} 1 - \|\tilde{x}\|^2, & \|\tilde{x}\| < 1 \\ 0, & \text{otherwise} \end{cases}$$

Where:

$$\tilde{x} = \begin{bmatrix} \frac{x - x_c}{W} \\ \frac{y - y_c}{H} \end{bmatrix}$$

We can compare the histograms using: Correlation, Intersection, etc.

Given a histogram template  $H_0$  and location  $x_{t-1}$  in frame  $I_{t-1}$ , search neighborhood in image  $I_t$  to find window with matching histogram.



This technique is more resilient to changes in object pose and/or scale.

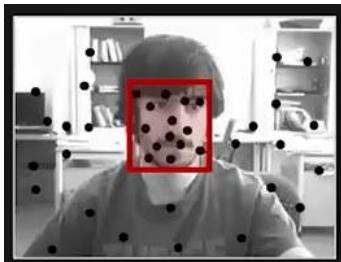
## OBJECT TRACKING BY FEATURE DETECTION

**At frame 1:**

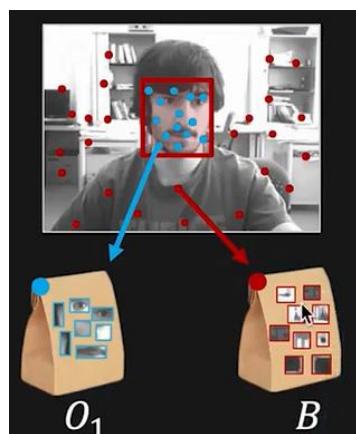
1. User selects a bounding box  $W_1$  as object/target.



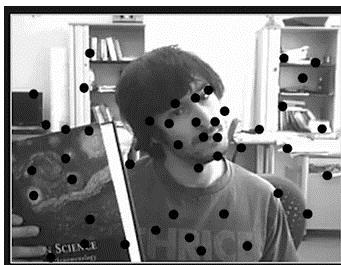
2. Compute SIFT (or similar) features for the frame.



3. Classify features within the box as object and assign them to set  $O_1$ .
4. Classify remaining features as the background and assign them to set  $B$ .

**At frame  $t$ :**

1. Compute SIFT features and SIFT descriptors  $\{v_1, \dots, v_k\}$  for frame  $I_t$ .



2. For each feature and corresponding descriptor  $v_i$ :
  - a. Compute distance  $d_O$  between  $v_i$  and the best match in the object set  $O_{t-1}$ .
  - b. Compute distance  $d_B$  between  $v_i$  and the best match in the background set  $B$ .

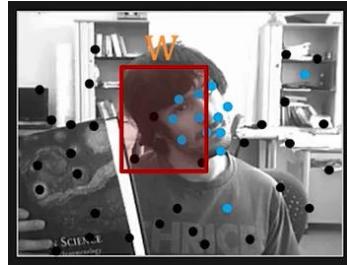
$$\text{c. } C(v_i) = \begin{cases} +1, & \text{if } \frac{d_o}{d_B} < 0.5 \ (\text{$v_i$ may belong to object}) \\ -1, & \text{otherwise} \ (\text{$v_i$ doesn't belong to object}) \end{cases}$$

$C(v)$  is a confidence value.

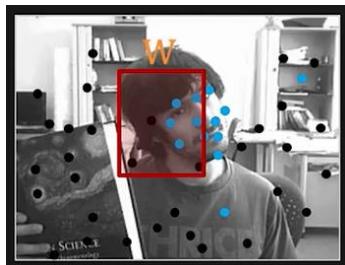
3. For each **Search Window  $W$** :

- a. Compute  $\varphi(W) = \sum C(v_i)$  for all features  $v_i$  inside  $W$ .
- b. Compute a heuristic  $\tau(W, W_{t-1})$  that penalizes large deviations from previous location, size and shape  $W_{t-1}$ .
- c. Compute **Match Score**:

$$\mu(W) = \varphi(W) - \tau(W, W_{t-1})$$



4. Select window  $W_t$  with the best match score as new object location.



5. Update object appearance model:

$$O_t = O_{t-1} \cup \{v_i\} \forall v_i \text{ inside } W_t \text{ such that } C(v_i) = +1.$$

## OBJECT TRACKING PROCEDURE (USING ADVANCED TECHNIQUES)

In general, the object tracking procedure is composed of 4 main modules:

- **Target initialization/object detection:** initial set of object detections is created. This is typically done by taking a set of bounding box coordinates and using them as inputs for the network. The idea is to draw bounding box of the target in the initial frame of the video and the tracker must estimate the target's position in the remaining frames in the video.
- **Appearance modeling:** learning the visual appearance of the object by using (deep) learning techniques. In this phase, the model learns the visual features of the object while in motion, various view-points, scale, illuminations etc.
- **Motion estimation:** the objective of motion estimation is learning to predict a zone where the target is most likely to be present in the subsequent frames.
- **Target positioning:** motion estimation predicts the possible region where the target could be present, thus, yielding an area to search to lock down the exact location of the target.

It is usually the case that tracking algorithms don't try to learn all the variations of the object. Hence, most of the tracking algorithms are much faster than regular object detection.

## TYPES OF ADVANCED TRACKING ALGORITHMS

We can classify object trackers according to whether they are based on **automatic object detection or manual**, whether they **track a single object or capable of tracking multi objects** and whether they operate **online or offline**.

### *Detection Based*

Here the consecutive video frames are given to a **pretrained object detector** that forms a **detection hypothesis** which in turn is used to **form tracking trajectories**.

- It is more popular because new **objects are detected and disappearing objects are terminated automatically**.
- In these approaches, the **tracker is used for the failure cases of object detection**.
- In another approach, object detector is run every  $n$  frames and the remaining predictions are done using the tracker.
- Suitable approach for tracking for a long time.

### *Detection Free*

This technique **requires manual initialization** of a fixed number of objects in the first frame. It then localizes these objects in the subsequent frames.

- Cannot deal with the case where new objects appear in the middle frames.

### *Single Object Tracking*

Here only a **single object is tracked** even if the environment has multiple objects in it. The object to be tracked is determined by the initialization in the first frame.

### *Multi Object Tracking*

Here **all the objects present in the environment are tracked over time**. If a detection based tracker is used it **can even track new objects that emerge in the middle of the video**.

### *Offline Trackers*

Those trackers are used when you must **track an object in a recorded stream**. For example, if you have recorded videos of a soccer game of an opponent team which needs to be analyzed for strategic analysis. In such case, you can not only use the past frames but also **can use future frames to make more accurate tracking predictions**.

### *Online Trackers*

**Online trackers** are used where **predictions are available immediately** and hence, they can't use future frames to improve the results.

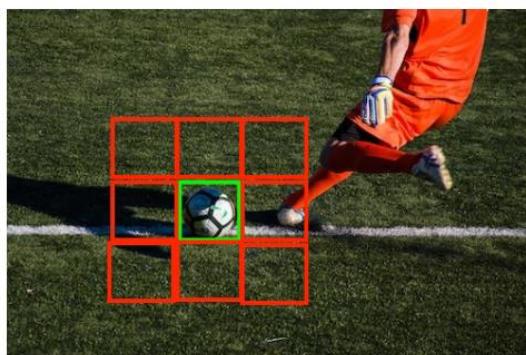
### *Offline Learning Trackers*

The **training** of these trackers only happen **offline**. As opposed to online learning trackers, these trackers **don't learn anything during run time**. We can train a tracker to identify persons and then these trackers can be used to continuously track all the persons in a video stream. Those are so called **Pre-trained Trackers**.

### *Online Learning Trackers*

Those trackers typically learn about the object to track using the initialization frame and few subsequent frames, making these trackers more general because you can just draw a bounding box around any object and track it. For example, if you want to track a person with red shirt in the airport, you can just draw a bounding box around that person in 1 or few frames. The tracker would learn about the object using these frames and would continue to track that person.

In online learning trackers, **Center Red Box** is specified by the user, it is taken as the **positive example** and all the boxes surrounding the object are taken as **negative class** and a classifier is trained which learns to distinguish the object from the background.



## SELF-SUPERVISED

We'll make an introduction to self-supervised learning. First, we'll define the term and talk about its importance in machine learning. Then, we'll present some examples of self-supervised learning and some limitations.

### Preliminaries

Over the past few years, the field of machine learning has revolutionized many aspects of our life with applications ranging from self-driving cars to predicting deadly diseases. A crucial factor in this tremendous progress is the availability of massive amounts of carefully labeled data.

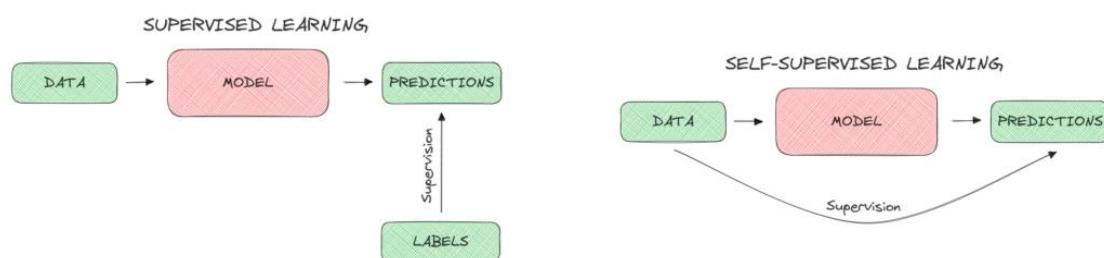
**However, it is quite clear that there is a limit on how far the field of machine learning can go with supervised learning and labeled data.** For example, there are many tasks where it is difficult or expensive to annotate a lot of data like translation for low-resource languages. **Therefore, to bring AI closer to human-level intelligence we should focus on methods that include training a model without annotations.** One of the most promising methods is self-supervised learning.

### Definition

The motivation behind self-supervised learning is to first learn general feature representations using unlabeled data and then fine-tune these representations on downstream tasks using a few labels. The question that arises is how to learn these useful representations without knowing the labels.

**In self-supervised learning, the model is not trained using a label as a supervision signal but using the data itself.** For example, a common self-supervised method is to train a model to predict a hidden part of the input given an observed part of the input.

Below, we can see diagrammatically how a supervised (left) and a self-supervised (right) model works:



### Importance

More and more self-supervised models are proposed nowadays that aim to substitute purely supervised learning models. Self-supervision is very important to the progress of machine learning for a number of reasons.

First, self-supervised learning **reduces the necessity of annotating a lot of data saving time and money.** Also, it enables the **use of machine learning in areas where annotating is difficult or**

**even impossible.** Another factor that illustrates the importance of self-supervision is the **vast amount of unlabeled data that are available everywhere**. Due to the extensive use of the internet and social media, there is a huge amount of images, videos, and audio clips that can be easily used to train self-supervised machine learning models.

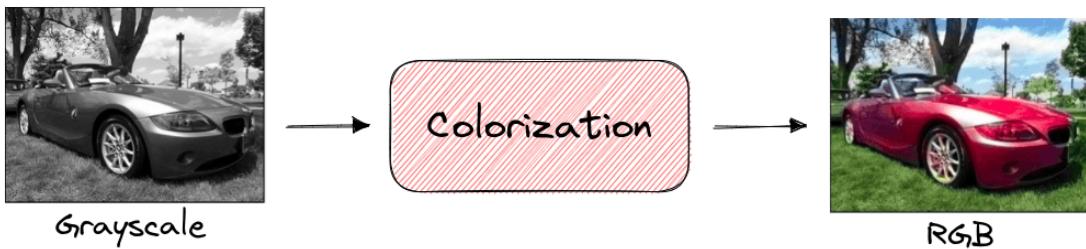
## EXAMPLES

Now, let's present some examples of self-supervised learning to better illustrate how it works.

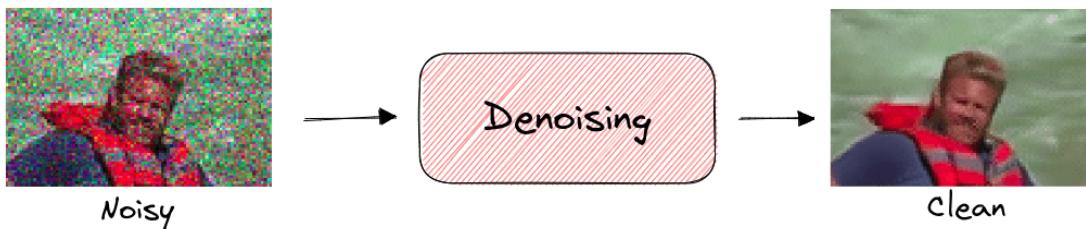
### Visual

A lot of self-supervised methods have been proposed for learning better visual representations. As we'll see below, there are many ways that we can manipulate an input image so as to generate a pseudo-label.

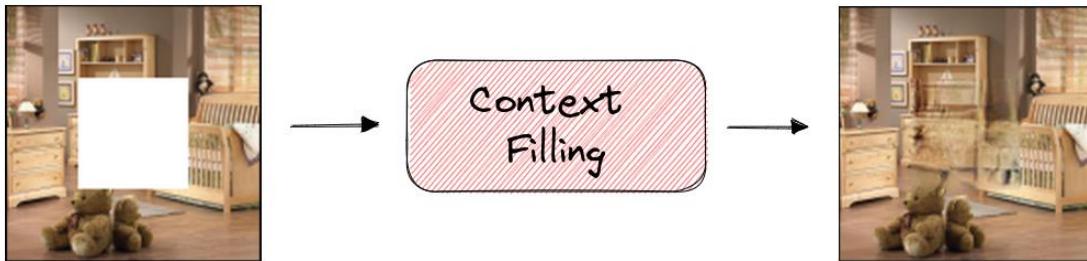
In [image colorization](#), a self-supervised model is trained to color a grayscale input image. So, during training, there is no need for labels since the same image can be used in its grayscale and RGB format. After training, the learned feature representation has captured the important semantic characteristics of the image and can then be used in other downstream tasks like classification or segmentation:



Another task that is common in self-supervision is [denoising](#) where the model learns to recover an image from a corrupted or noisy version. The model can be trained in a dataset without labels since we can easily add any type of [image noise](#) in the input image:



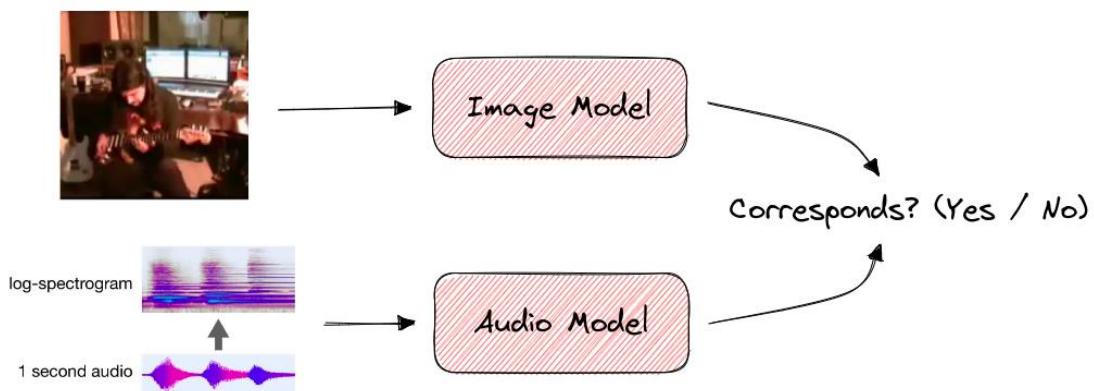
In [image inpainting](#), our goal is to reconstruct the missing regions in an image. Specifically, the model takes as input an image that contains some missing pixels and tries to fill these pixels so as to keep the context of the image consistent. Self-supervision can be applied here since we can just crop random parts of every image to generate the training set:



### *Audio-Visual*

**Self-supervised learning can also be applied in audio-visual tasks like finding [audio-visual correspondence](#).**

In a video clip, we know that audio and visual events tend to occur together like a musician plucking guitar strings and the resulting melody. **We can learn the relationship between visual and audio events by training a model for audio-visual correspondence.** Specifically, the model takes as input a video and an audio clip and decides if the two clips correspond to the same event. Since both the audio and visual modality of a video are available beforehand, there is no need for labels:



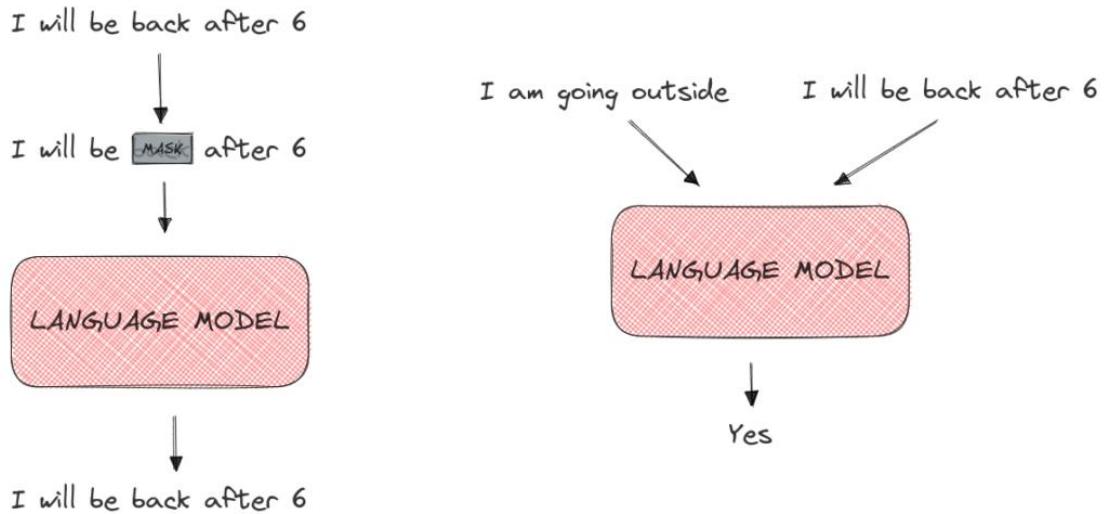
### *Text*

When training a language model it is very challenging to define a prediction goal so as to learn rich word representations. **Self-supervised learning is widely used in the training of large-scale language models like [BERT](#).**

To learn general word representations without supervision, two self-supervised training strategies are used:

- **MaskedLM** where we hide some words from the input sentence and train the language model to predict these hidden words.
- **Next Sentence Prediction** where the model takes as input a pair of sentences and learns their relationship (if the second sentence comes after the first sentence).

Below, we can see an example of the MaskedLM (left) and the Next Sentence Prediction (right) training objectives:



## LIMITATIONS

Despite its powerful capabilities, self-supervised learning presents some limitations that we should always take into account.

### *Training Time*

There is already a lot of controversy over the high amount of time and computing power that the training of a machine learning model takes with a negative impact on the environment. **Training a model without the supervision of real annotations can be even more time-consuming.** So, we should always compare this extra amount of time with the time it takes to just annotate the dataset and work with a supervised learning method.

### *Accuracy of Labels*

In self-supervised learning, we generate some type of pseudo-labels for our models instead of using real labels. **There are cases where these pseudo-labels are inaccurate hurting the overall performance of the model.** So, we should always check the quality of the generated pseudo-labels.

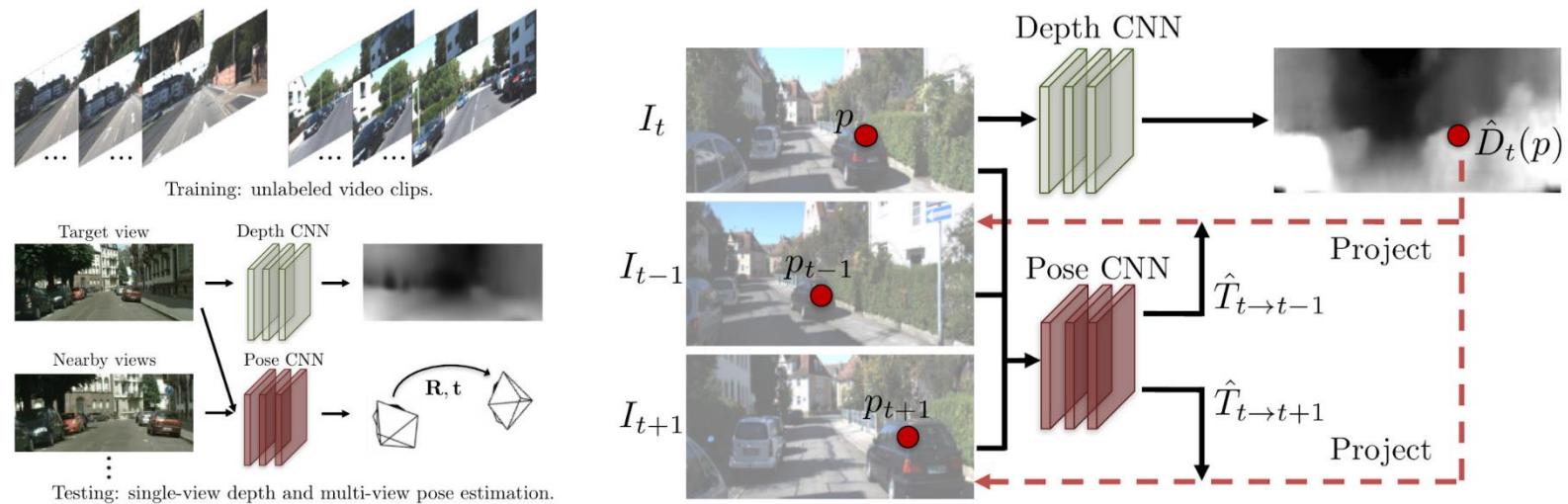
## Task-Specific Models

**Task-specific models** are self-supervised models that are specific to particular target tasks. This type of models is different from self-supervised models that are trying to learn generic representations independent of any downstream task.

### UNSUPERVISED LEARNING OF DEPTH AND EGO-MOTION

The first problem that we consider is unsupervised learning of depth and ego-motion.

The **input** at training time are **monocular videos** that can be downloaded from the internet or can be recorded by a camera mounted on top of a vehicle. The aim is then to train two neural networks (indicated in green and red in the following figure).



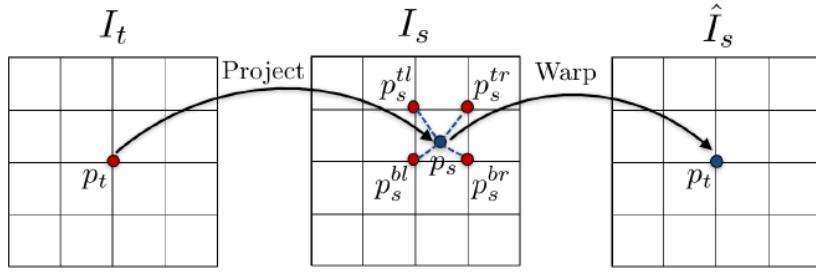
The neural network highlighted in green (Depth CNN) is a CNN that takes a single image as input and produces a dense depth map as output. Note that unlike the settings we have considered in this course so far, we do not have access to the ground truth depth map for this problem. The neural network highlighted in red (Pose CNN) is a CNN that takes in two adjacent frames as input and predicts the relative camera motion, consisting of the rotation matrix  $R \in \mathbb{R}^3$  and the translation vector  $t \in \mathbb{R}^3$  (i.e. six DoFs). The goal of the architecture is to **jointly predict the depth and relative pose from two or three adjacent video frames**.

The way this task is approached is heavily inspired by the way classical problems like optical flow are tackled. Also here, the photoconsistency assumption that has been used for optimizing the flow field is used to optimize the parameters of the Depth CNN as well as the Pose CNN. The model shown in figure above uses three adjacent frames as input: the “**target view**” at time  $t$  (indicated by  $I_t$ ) as well as two “**source views**” at times  $t - 1$  and  $t + 1$  (indicated by  $I_{t-1}$  and  $I_{t+1}$ , respectively). The Depth CNN predicts a depth  $\hat{D}_t(p)$  for every pixel location  $p$  in the target view. The Pose CNN predict the relative motion between (1) the target view  $I_t$  and the source view  $I_{t-1}$  (indicated by  $\hat{T}_{t \rightarrow t-1}$ ) and (2) the target view  $I_t$  and the source view  $I_{t+1}$  (indicated by  $\hat{T}_{t \rightarrow t+1}$ ). Note that each  $\hat{T}$  here consists of  $R$  and  $t$ . Because we now know every depth of every pixel location  $p$  in the target view, we can project it into 3D. Furthermore, because we know the

relative pose between the all three views (or frames), we can project point  $p$  in target image  $I_t$  onto  $I_{t-1}$  and  $I_{t+1}$ , based on the following equation (first arrow in the figure):

$$\tilde{p}_s = K((RD(\bar{p}_t)K^{-1}\bar{p}_t) + t)$$

the predicted depth  $D(\bar{p}_t)$  and the relative pose  $R$  and  $t$  depend of the parameters of the Depth CNN and the Pose CNN as well as on the input frames.  $K$  is the intrinsic matrix of the camera. Assuming the scene to be static, we therefore know the correspondence of pixel  $p$  between  $I_t$  and  $I_{t-1}$  as well as between  $I_t$  and  $I_{t+1}$ , which is given implicitly through the depth map  $\hat{D}_t$  and the relative pose  $\hat{T}_{t \rightarrow t-1}$  and  $\hat{T}_{t \rightarrow t+1}$ . Given this information, we can effectively warp the two source views  $I_{t-1}$  and  $I_{t+1}$  into the target image  $I_t$  (the second arrow in the figure).



Due to the fact that after the projection based on the equation above, the estimated location might not match an actual, discrete pixel location, bilinear interpolation of RGB color values of adjacent pixels is used during the warping process. After the warping, we can then compare the pixels of the original target view to the warped source views based on the photoconsistency loss:

$$\mathcal{L} = \sum_p |I_t(p) - \hat{I}_s(p)|$$

Note that the photoconsistency loss is computed for all warped source images  $\hat{I}_s$  and is then averaged across them. Assuming a static scene, Lambertian surfaces and photoconsistency, we would ideally expect the difference between the RGB pixel values to be zero, given that the predicted depth and the predicted relative pose is correct. However, if the predicted depth and the predicted pose are incorrect, then there will be a discrepancy between the RGB values of the compared pixels. This discrepancy - or photoinconsistency - can be used to back-propagate gradients to the parameters of the Depth CNN as well as the Pose CNN, in order to update these parameters. This is possible, because the projection as well as the bilinear interpolation are differentiable. All in all, we try to minimize the photoconsistency for a lot of monocular sequences of frames without requiring ground truth for depth or relative pose.

The particular architecture that is used for the model by Zhou is a U-Net (DispNet) architecture with a multi-scale prediction loss. This means that the depth as well as the pose are predicted at multiple scales and that the source views are warped at multiple scales. This improves gradient flow and thereby also optimization. The final objective includes the before mentioned photoconsistency loss, as well as a smoothness and an explainability loss. It is crucial to have a smoothness loss on the disparity map for the same reasons that we had to introduce

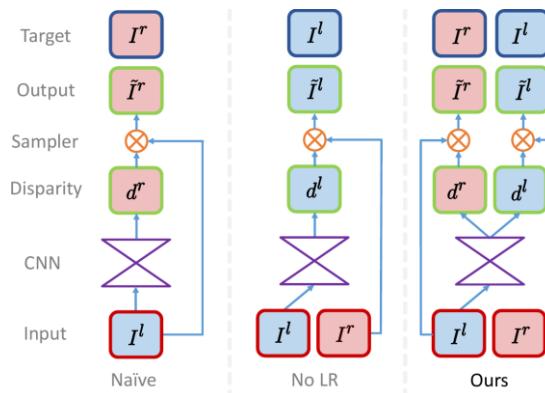
smoothness constraints in methods that were covered earlier in the course, i.e. stereo and optical flow estimation.

Already back in 2017, the performance of the self-supervised method by Zhou was nearly on par with depth- or pose-supervised methods. Compared to depth-supervised methods, which use LiDAR scanners for ground truth that is not available for the whole image, the self-supervised model is able to produce results for all regions of a given image. Nevertheless, a downside of the model is that it assumes static scenes, so it can fail in the presence of dynamic objects.

## UNSUPERVISED DEPTH ESTIMATION FROM STEREO

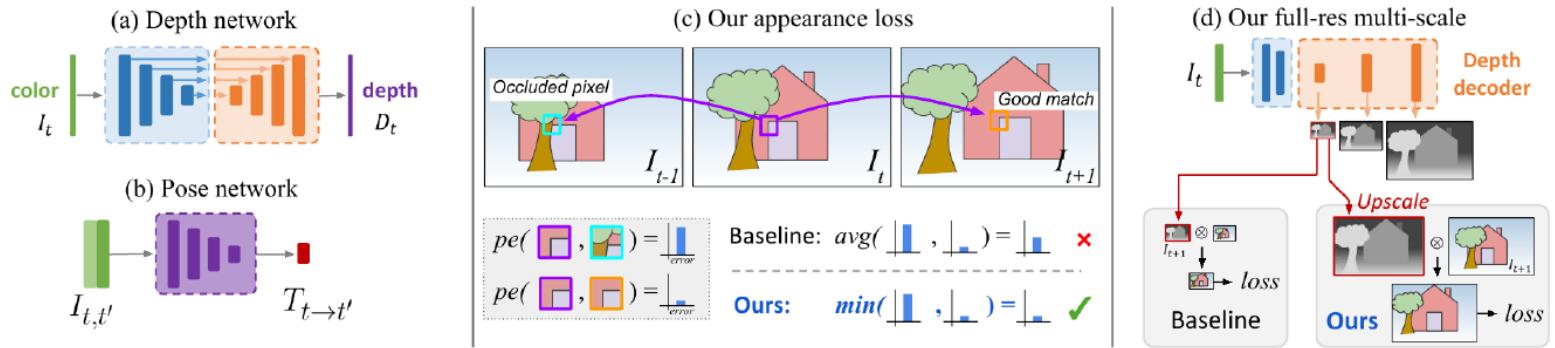
Another way of training a monocular depth estimation network in a self-supervised fashion is based on stereo (i.e. based on images that have been taken at the same time from adjacent viewpoints). The advantage is that one can get sharper results and that dynamic scenes are of no concern, because images have been taken simultaneously. In their paper from 2017, Godard, Aodha and Brostow propose an idea applied to the pair of stereo images instead of consecutive images. They proposed three different kinds of models as shown in Fig. 141. The “naive” model (in the left in the figure) takes the left input image (i.e.  $I^l$ ) and runs it through a CNN that produces a disparity map (i.e.  $d^r$ ). Because the left image is in the end compared to the right target image (i.e.  $I^r$ ), it has to be first represented in the target image coordinate frame. This is due to the fact that the “sampler” (i.e. the bilinear interpolation) can only interpolate points in one direction. In other words, one can only warp an image into another image for which the disparity map is given. Therefore, the left image  $I^l$  has to be warped using the disparity map that is defined in the right image frame (i.e.  $d^r$ ). However, our aim is not to predict the disparity map for the right target image  $d^r$ .

The “**No LR**” model (in the middle in the figure) tries to correct for this. Here, the left image  $I^l$  is run through the CNN to produce the left disparity map  $d^l$ . Using this disparity map the right image (i.e.  $I^r$ ) is then warped into the coordinate frame of the left image (i.e.  $I^r \rightarrow \tilde{I}^l$ ) so that it can be compared to the original left image  $I^l$  based on a photoconsistency loss. While this approach works better, it suffers from unwanted artefacts.



The approach that worked best (in the right in the figure) is to run the left image (i.e.  $I^l$ ) through a monocular depth estimation network that outputs not only the disparity map for the left image (i.e.  $d^l$ ), but also for the right image (i.e.  $d^r$ ). Based on both these disparity maps, we can then

take the left image and warp it to the right image coordinate frame (i.e.  $I^l \rightarrow \hat{I}^r$ ) based on  $d^r$ . Similarly, we can take the right image and warp it to the left image coordinate frame (i.e.  $I^r \rightarrow \hat{I}^l$ ) based on  $d^l$ . The advantage of this is that we can now apply a consistency constraint between the two disparity maps  $d^l$  and  $d^r$ , which helps to eliminate outliers and improves training. Therefore, the losses that are used in this method are a photoconsistency loss, a disparity smoothness loss as well as a disparity left-right consistency loss.



In a followup work, the same authors more generally looked into self-supervised monocular depth estimation methods and they introduced some improvements. For example, a per-pixel minimum photoconsistency loss was introduced to handle occlusions. The baseline approach averages the photoconsistency loss across the comparisons of the target view to the warped source view(s). This leads to a relatively high photoconsistency loss in the case that in one of the source views a pixel is occluded. In contrast, the method proposed by the authors takes the minimum photoconsistency loss over the comparisons. Because the model assumes three consecutive frames as input, the occluded pixel in one of the source views will likely not be occluded in the other source view. Therefore, the model effectively ignores the occlusion and still leads to a small overall photoconsistency loss. Another contributions of the authors is to compute all losses at the resolution of the input images. To this end the multi-scale predictions of the model are scaled up using bilinear interpolation before computing the losses. The authors have found that doing this can avoid texture-copy artefacts. Lastly, the authors introduced an auto-masking loss which helps to ignore input images in which the camera does not move, as for this kind of input the problem becomes ill-posed and cannot be solved, which harms the training of the network. By implementing the above mentioned improvements, the authors could produce significantly sharper depth boundaries and more details.

## Pretext Tasks

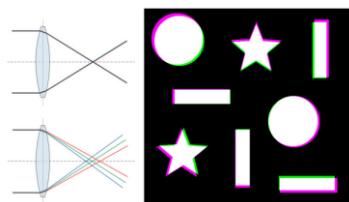
Before we have focused on self-supervised models which are tailored to a specific downstream task. This section covers the learning of more general neural representations using self-supervision. To achieve this, we must define an **auxiliary task** - or **pretext task**. An example of this is predicting the rotation of an image which has been randomly rotated. By doing this, the hope is that we learn useful features which will be helpful for some other downstream task. This kind of pre-training can be performed on lots of unlabeled data which in most cases is easily accessible via the internet. After the pre-training has taken place, we can then remove the last layers of the pre-trained model and instead add new layers to perform the target task (e.g. classification). We then train the whole network to perform the target task. Because pre-training has (hypothetically) already produced strong and useful features, we now need much less data to train the model to perform the target task.

The following sections introduce some pretext tasks that have been proposed in the literature. All of these might appear quite hand-engineered. This is due to the fact that there is not yet a theoretical basis for designing pretext tasks. As a result, the design of pretest tasks still heavily relies on empirical data which indicates that they work well.

### VISUAL REPRESENTATION LEARNING BY CONTEXT PREDICTION

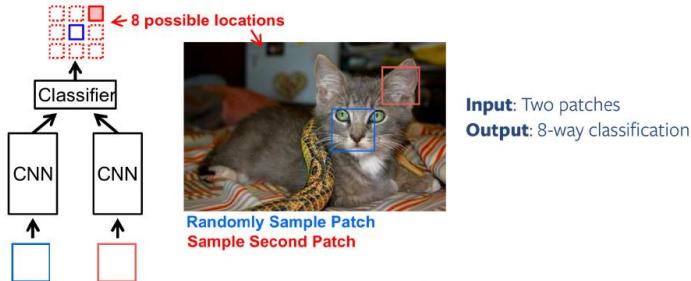
In this task the goal is, given two image patches, to predict their relative location in the image. The locations of the patches per image are discretized into eight possible patch locations. This makes the task an eight-way classification task. The hope underlying this approach is that the model learns to recognize objects and their parts. By design, the task forces the model to learn something about the composition of objects.

Importantly, special care has to be taken in order to avoid that the network takes trivial short cuts to solve the task. For example, it has become clear in practice, that the network simply uses edge continuity. To prevent these short cuts, the authors proposed to leave gaps between the center patch and its eight neighboring patches and to spatially jitter the locations of the eight neighboring patches. However, the authors realized that this does not fully solve the problem. They noticed that the network was able to predict the absolute location of randomly sampled patch in an image, from which the relative location of patches can be inferred easily. The authors found out, that the model was able to do so due to the so called aberration of camera lenses. Aberration describes the systematic shift of color channels with respect to the image location.



Based on the information of this systematic color shift alone, the model was able to predict the absolute location of patches. A solution to this short cut is to randomly drop color channels or to project the whole image to grayscale.

With these measures to prevent trivial short cuts implemented, the model was actually able to learn useful features to perform downstream tasks such as object detection. The authors found that pre-training an R-CNN on this context prediction task without labels followed by fine-tuning the fully-connected layers for classification on a small data set led the resulting model to perform much better, compared to a model without pre-training.



Surprisingly, the same approach has also been shown to work for surface-normal estimation, which is a completely different task.

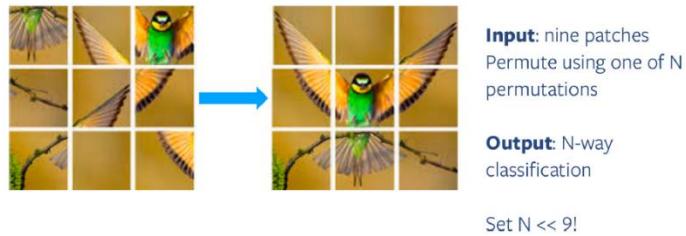
The authors also investigated the visual representations of what the network has learned. To this end, they used nearest neighbor's search in the feature space of the last fully-connected layers of the architecture. The authors compared results from (1) random initialization of the network, (2) form AlexNet trained on ImageNet and (3) their pre-trained method. While random initialization (as expected) does not lead to meaningful results, the results of the pre-trained network are as semantically meaningful, as the results of the network trained on ImageNet.

### VISUAL REPRESENTATION LEARNING BY SOLVING JIGSAW PUZZLES

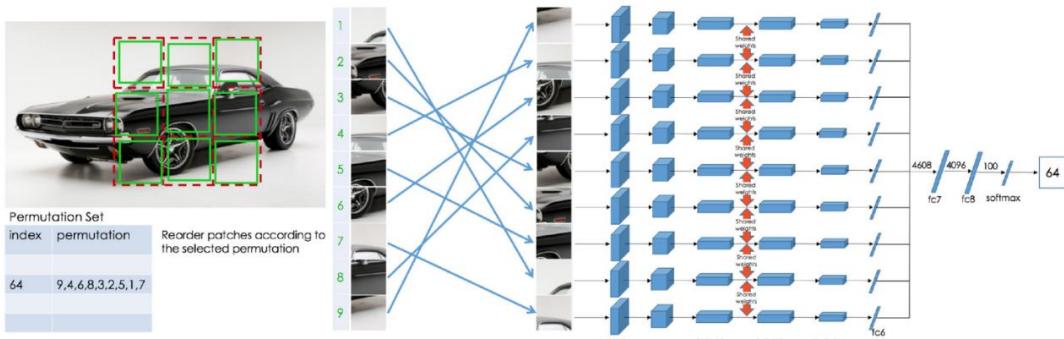
In a jigsaw puzzle pre-test task the idea is to randomly permute a version of the image that is separated into a  $3 \times 3$  array of patches (following figure) and train a network to recover the configuration of patches that corresponds to the original image. Because theoretically, there are very many (i.e.  $9!$ ) possible configurations, the authors chose to restrict the number to 1000 possible permutations. This makes the problem a 1000-way classification task. In addition, these permutations were chosen based on the Hamming distance to increase the level of difficulty (i.e. the authors chose particularly hard permutations in order to make the model learn better representations).

Also for this jigsaw puzzle task, one has to make sure to prevent the model from learning trivial short cuts which may be useful for solving the pre-text task, but not for solving the downstream target task.

- **Low-level statistics:** The first such short cut is using low-level statistics. Adjacent patches in an image often share similar low-level statistics, i.e. mean and variance. A solution to avoid this kind of short cut is to normalize the patches based on their mean and variance.
- **Edge continuity:** This is the same short cut as mentioned in the previous section.
- **Chromatic Aberration:** This is the same short cut as mentioned in the previous section.



By inspecting the visual representations that the model learns, the authors found that the model including these counter-measures for short cut learning indeed was able to learn meaningful representations throughout the layers of the architecture. The authors further compared their pre-trained model with fine-tuning for the PASCAL VOC challenge (i.e. classification, detection and segmentation) to other established methods such as fully-supervised pre-training on ImageNet and fine-tuning on PASCAL VOC, which was the standard approach at the time. They found that the performance of their model was nearly at par with the standard approach. This shows that self-supervised pre-training on the jigsaw puzzle task is a powerful method which produces results that can keep up quite well with standard, fully-supervised methods, at least for some tasks.



## EXAMPLE QUESTION

- A key problem in designing pretext tasks for self-supervised learning is the tendency of the network to prioritize trivial shortcuts over meaningful learning. Considering the problem of learning by solving jigsaw puzzles, what are the three main shortcuts exploited by the network, and the solutions proposed to overcome these?

### Solution:

- Low level statistics:** Adjacent patches include similar low-level statistics (mean and variance). **Solution:** normalize patch mean and variance.
- Edge continuity:** A strong cue to solve Jigsaw puzzles is the continuity of edges. **Solution:** select 64x64 pixel tiles randomly from 85x85 pixel cells.
- Chromatic Aberration:** Chromatic aberration is a relative spatial shift between color channels that increases from the images center to the borders. **Solution:** use grayscale images or spatial jitter each color channel by few pixels.

b. Consider the following two pretext tasks:

- a. Predicting if an image has been horizontally flipped.
- b. Predicting if an image has been vertically flipped.

When trained on a dataset of images from the internet, which of the two pretext tasks would you expect to lead to better self-supervised representations, and why?

**Solution:**

Predicting if a natural image has been horizontally flipped is an ill-posed problem, with the exception of certain specific kinds of images (eg. images containing text). For most natural images, there should be no underlying feature that can be used to determine if the image has been mirrored. If this pretext task can be learned by a self-supervised model, it is likely that the model is exploiting some shortcut, and therefore the representations learned are less likely to be useful.

On the other hand, predicting vertical flips is a reasonable pretext task for images from the internet. A model that can accurately predict if an image has been flipped vertically based on whether objects in the image are in their 'canonical pose' (i.e., using relative position cues of what is usually at the top and bottom of an object). Features learned for this task could generalize to other image-based tasks.

c. If the same two pretext tasks from the previous question were applied to satellite images, which of the two would you now expect to lead to better representation learning?

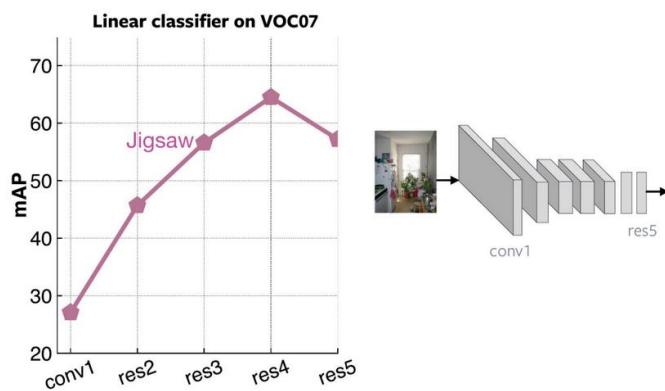
**Solution:**

Satellite images typically do not have strong spatial biases/structure like images from the internet. The images have a lot of texture and not many well-defined objects. Furthermore, the objects do not have a 'canonical pose'. Both kinds of image flipping are therefore unlikely to lead to useful representations for satellite images, since the problem is again ill-posed.

## CONTRASTIVE LEARNING

We'll introduce the area of **contrastive learning**. First, we'll discuss the intuition behind this technique and the basic terminology. Then, we'll present the most common contrastive training objectives and the different types of contrastive learning.

The problem with pretext tasks is that for learning the majority of parameters during pre-training, we are considering a task that is completely decoupled from the downstream target task, i.e. solving jigsaw puzzles (pretext task) and image classification (target task). Therefore, we can only hope that the pretext task and the target task are somewhat aligned. In fact, it can be observed that the performance of image classification heads that are attached to models that have been pre-trained on a jigsaw puzzle task **saturates for deeper layers**.



This is somewhat counter-intuitive, because one would think that applying the classification heads to deeper layers in the pre-trained architecture should lead to better results, because **deeper layers usually contain more semantically meaningful features compared to early layers**. This indicates that the **last layers of the pre-trained model have specialized in solving the pretext task**, i.e. the last layers are very specific to solving context prediction problems such as jigsaw puzzles (but they don't contain the semantic knowledge required for solving the image classification task).

## Intuition

First, let's talk about the intuition behind contrastive learning. Below, we can see a traditional game that many kids play:



**The goal of this game is to look at the pictures on the right side and search for an animal that looks like the one on the left side.** In our case, the kid has to search for a picture of a dog among the four pictures on the right. First, the kid has to compare each one of the four animals with a dog and then conclude that the bottom left image depicts a cat.

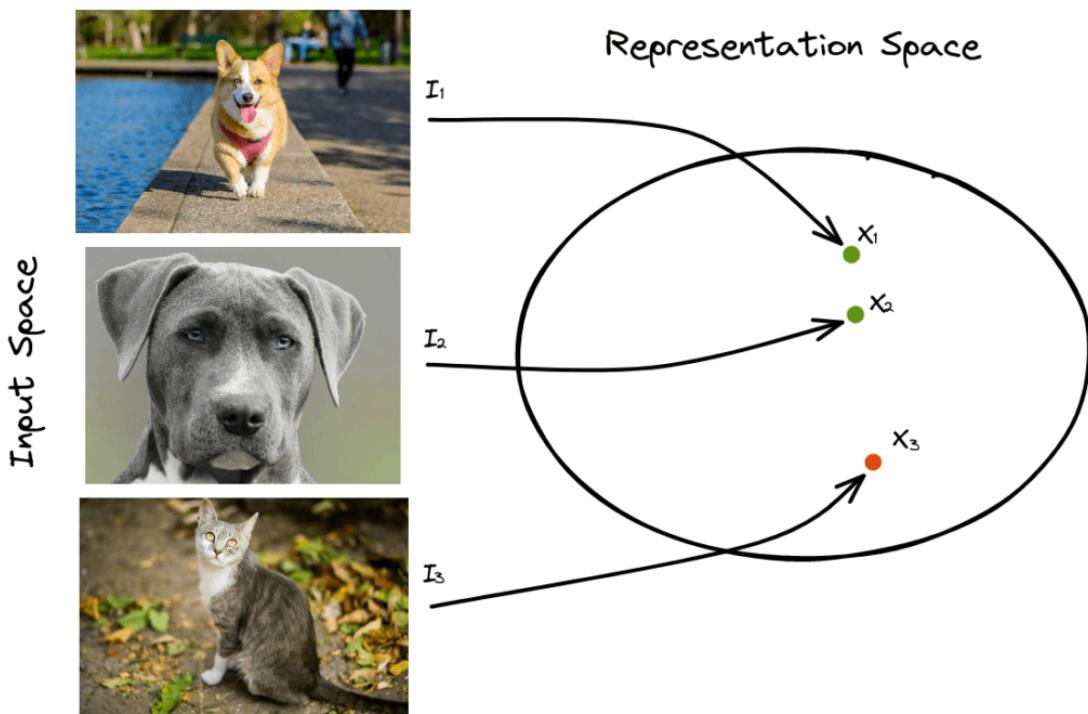
According to many surveys, kids learn more easily new concepts in this way than reading a book about animals. But why does this method work better?

**It turns out that it is easier for someone with no prior knowledge, like a kid, to learn new things by contrasting between similar and dissimilar things instead of learning to recognize them one by one.** At first, the kid may not be able to identify the dog. But after some time, the kid learns to distinguish the common characteristics among dogs, like the shape of their nose and their body posture.

## Terminology

Inspired by the previous observations, contrastive learning aims at learning low-dimensional representations of data by contrasting between similar and dissimilar samples. Specifically, it tries to bring similar samples close to each other in the representation space and push dissimilar ones to be far apart using the Euclidean distance. At the same time, learned features should be invariant to nuisance factors, such as the location of the object, lighting conditions or color.

Let's suppose that we have three images  $I_1$ ,  $I_2$  and  $I_3$ . The first two images depict a dog, and the third image depicts a cat, and we want to learn a low-dimensional representation for each image  $x_1$ ,  $x_2$  and  $x_3$ ):

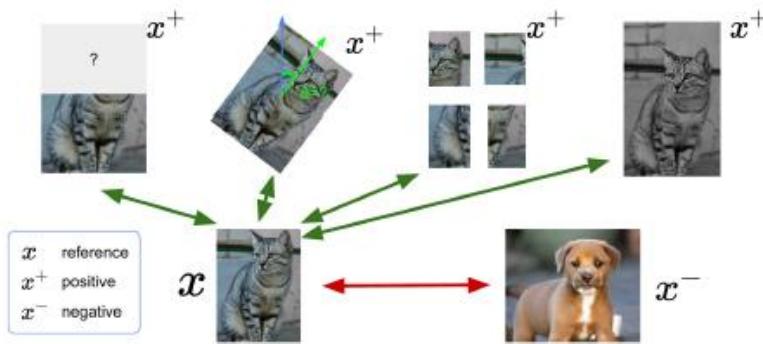


In contrastive learning, we want to **minimize the distance between similar samples and maximize the distance between dissimilar samples**. In our example, we want to minimize the distance  $d(x_1, x_2)$  and maximize the distances  $d(x_1, x_3)$  and  $d(x_2, x_3)$  where  $d()$  is a metric function like Euclidean.

The sample that is **similar to the anchor sample ( $I_1$ ) is defined as a positive sample ( $I_2$ )** and the one that is **dissimilar as negative ( $I_3$ )**.

Or choose some score function  $s(\cdot, \cdot)$  and we want to learn encoder  $f$  that yields a high score for positive pairs  $(x, x^+)$  and low score for negative pairs  $(x, x^-)$ , such that:

$$s(f(x), f(x^+)) \gg s(f(x), f(x^-))$$



## Training Objectives

The most important part of contrastive learning is the training objective that guides the model into learning contrastive representations. Let's describe the most common objectives.

### CONTRASTIVE LOSS

Contrastive loss is one of the first training objectives that was used for contrastive learning. It takes as input a pair of samples that are either similar or dissimilar, and it brings similar samples closer and dissimilar samples far apart.

More formally, we suppose that we have a pair  $(I_i, I_j)$  and a label  $y$  that is equal to 0 if the samples are similar and 1 otherwise. To extract a low-dimensional representation of each sample, we use a Convolutional Neural Network  $f$  that encodes the input images  $I_i$  and  $I_j$  into an embedding space where  $x_i = f(I_i)$  and  $x_j = f(I_j)$ . The contrastive loss is defined as:

$$L = (1 - y) \cdot \|x_i - x_j\|^2 + y \cdot \max(0, m - \|x_i - x_j\|^2)$$

where  $m$  is a hyperparameter, defining the lower bound distance between dissimilar samples.

If we analyze in more detail the above equation, there are two different cases:

- If the samples are similar ( $y = 0$ ), then we minimize the term  $\|x_i - x_j\|^2$  that corresponds to their Euclidean distance.

- If the samples are dissimilar ( $y = 1$ ), then we minimize the term  $\max(0, m - \|x_i - x_j\|^2)$  that is equivalent to maximizing their Euclidean distance until some limit  $m$ .

## TRIPLET LOSS

An improvement of contrastive loss is [triplet loss](#) that outperforms the former by using triplets of samples instead of pairs.

Specifically, it takes as input an anchor sample  $I$ , a positive sample  $I^+$  and a negative sample  $I^-$ .

During training, the loss enforces the distance between the anchor sample and the positive sample to be less than the distance between the anchor sample and the negative sample:

$$L = \max(0, \|x - x^+\|^2 - \|x - x^-\|^2 + m)$$

When we train a model with the triplet loss, we require fewer samples for convergence since we simultaneously update the network using both similar and dissimilar samples. That's why triplet loss is more effective than contrastive loss.

## INFONCE LOSS

One of the contrastive learning losses is the **InfoNCE** loss:

$$\mathcal{L}_N = -E_X \left[ \log \frac{f_k(x_{t+k}, c_t)}{\sum_{x_j \in X} f_k(x_j, c_t)} \right]$$

where the **numerator** is essentially the **output of a positive pair**, and the **denominator** is the **sum of all value of positive and negative pairs**. Ultimately, this simple loss forces the positive pairs to have a greater value (pushing the log term to 1 and hence less to 0) and negative pairs further apart.

If we assume that we have one reference ( $x$ ), one positive ( $x^+$ ) and  $N - 1$  negative ( $x_j^-$ ) examples. We can show the score as a **multi-class cross entropy loss function**:

$$\mathcal{L}_N = -E_X \left[ \log \frac{\exp(s(f(x), f(x^+)))}{\exp(s(f(x), f(x^+))) + \sum_{j=1}^{N-1} \exp(s(f(x), f(x_j^-)))} \right]$$

Note the use of the scoring function  $s(\cdot, \cdot)$  in the equation above. Its negative (i.e.  $-\mathcal{L}$ ) is a **lower bound on the mutual information** between  $f(x)$  and  $f(x^+)$ :

$$MI[f(x), f(x^+)] \geq \log(N) - \mathcal{L}$$

Importantly, the larger the negative sample size ( $N$ ), the tighter the bound, which is why we need a large  $N$  for this to work. The key idea behind this loss is **to maximize the mutual information between features extracted from multiple “views”**, which forces the features to capture information about higher-level factors. In other words, by minimizing the loss, we maximize the mutual information between multiple views of the reference image.

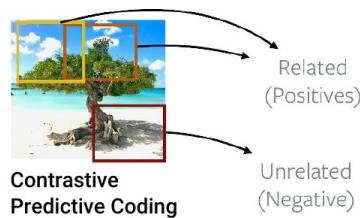
## Design Choices for Contrastive Learning

There are a number of design choices for these contrastive methods, that one can consider. The first design choice is concerned with the **scoring function  $s(\cdot, \cdot)$** . The most common choice for the scoring function is the cosine similarity:

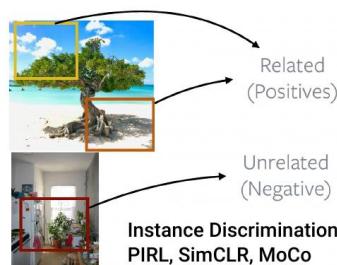
$$s(f_1, f_2) = \frac{f_1^T f_2}{\|f_1\| \|f_2\|}$$

Which is the inner product between the features divided by their norm. As a side note, here we are looking at a “**Siamese network**” because the features are computed with the same network.

The second design choice is concerned with the choice of the **positive and negative examples - or views**. In **Contrastive Predictive Coding** the **examples are chosen from the same image in a way that related (positive) examples/views are taken from nearby image regions, while unrelated (negative) examples/views are taken from further away image regions**.



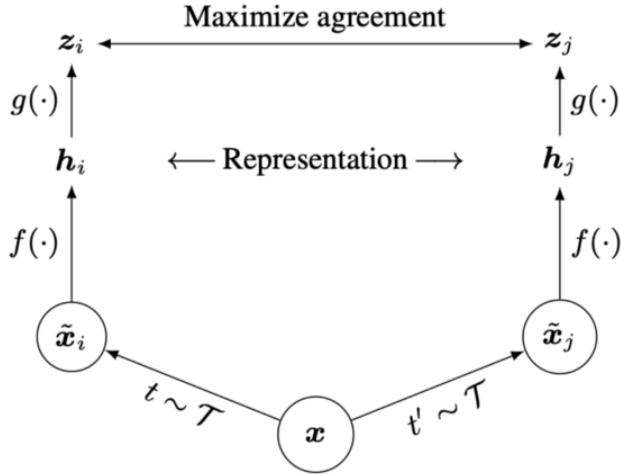
A method that is more common nowadays is so called **Instance Discrimination**. Here **all related examples come from a single image and all unrelated examples come from a different image**. This method assumes that the image from which the related examples are taken is a simple image that depicts only a single object.



The third design choice is concerned with the **augmentations** that are used on the related examples. Possible augmentations are to crop, resize or flip images, to apply rotation or cutout, to drop or jitter colors, to apply Gaussian noise or blur or to apply other kinds of filters.



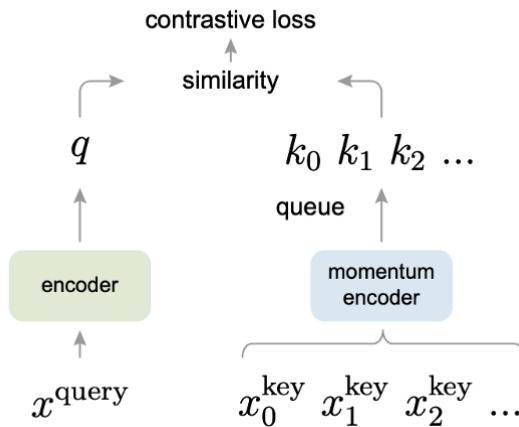
*Simple Framework for Contrastive Learning of Visual Representations  
 (SimCLR)*



**SimCLR** is the first paper to suggest using contrastive loss for self-supervised image recognition learning through image augmentations.

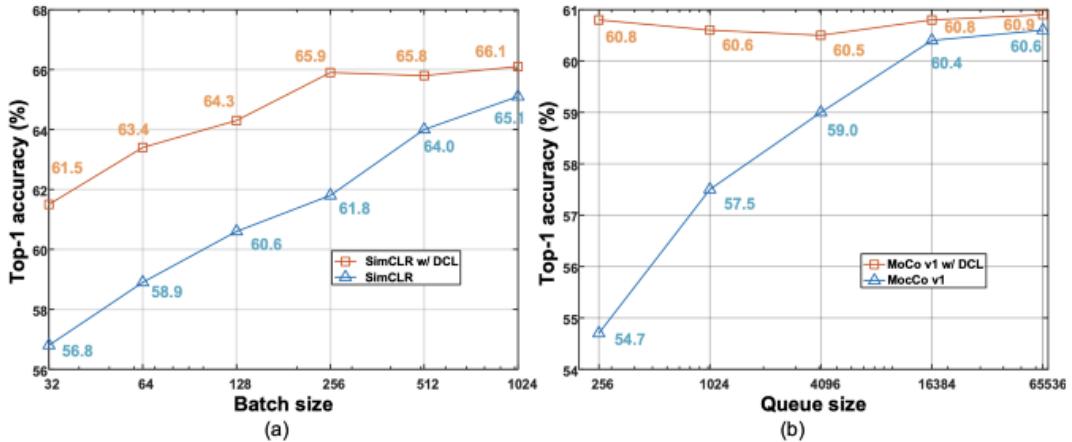
By generating positive pairs by doing data augmentation on the same image and vice versa, we can allow models to learn features to distinguish between images without explicitly providing any ground truths.

*Momentum Contrast (MoCo)*



The previous InfoNCE loss is proposed on a mini-batch of one positive and a number of negatives. He et al. extended this concept by portraying the contrastive learning as analogous to learning to **match the best key with a given queue**. The intuition led to the foundation of **momentum contrast (MoCo)**, which is essentially a dictionary/memory network of key and values with key stored across multiple batches and slowly eliminating the oldest batch in a queue-like manner. This allows the training to be more stable as it is similar to a momentum where the change in keys is less drastic.

### *Decoupled Contrastive Learning (DCL)*



Previous papers in contrastive learning either required large batch sizes or a momentum mechanism. The recent paper **decoupled contrastive learning (DCL)** hope to change this by bringing a simple change to the original InfoNCE loss: simply **removing the positive pair from the denominator**.

While seemingly simple, **DCL actually allows better convergence** and ultimately formed an even better baseline compared to previous papers SimCLR and MoCo.

## Types of Learning

The idea of contrastive learning can be used in both supervised and unsupervised learning tasks.

### SUPERVISED

In this case, the label of each sample is available during training. So, **we can easily generate positive and negative pairs or triplets by just looking into the labels.**

However, generating all possible pairs or triplets requires a lot of time and computational resources. Also, in every dataset, there are many negative pairs or triplets that already satisfy the contrastive training objectives and give zero loss resulting in slow training convergence.

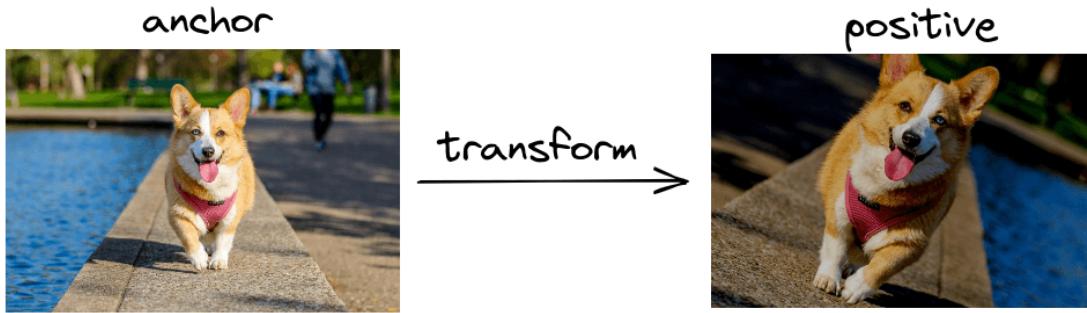
To deal with this problem, **we have to generate hard pairs and hard triplets, meaning that their loss value is high, i.e., similar pairs that are far apart and dissimilar pairs that are very close.**

Many hard negative mining methods have been proposed that usually look into the representation space for hard pairs and triplets using fast search algorithms. In natural language processing, a simple way of generating a hard negative pair of sentences is to add in the anchor sentence a negation word.

### UNSUPERVISED

When we don't have labeled samples, **self-supervised learning** is used where we exploit some property of our data to generate pseudo-labels.

A famous self-supervised framework for unsupervised contrastive learning is [SimCLR](#). Its main idea is to generate positive image pairs by applying random transformations in the anchor image like crop, flip and color jitter since these changes keep the label of the image unchanged:



### EXAMPLE QUESTION

- a. An important design choice in nearly all contrastive learning methods is the score function or similarity metric. Given two features  $f_1$  and  $f_2$ , write down the mathematical equation for the 3 following commonly used score functions:
- L1 (Manhattan) distance
  - L2 (Euclidean) distance
  - Cosine similarity

Rank the three functions based on how susceptible you think they are to outliers when used in high-dimensional feature spaces.

**Remark:** An outlier here refers to an example for which the features lie far away from the mean of the data distribution.

#### Solution:

Let  $f_k^i$  denote the  $i^{th}$  dimension of the vector  $f_k$ .

$$L_1 \text{ (Manhattan) distance: } s(f_1, f_2) = -\|f_1 - f_2\|_1 = -\sum_i |(f_1^i - f_2^i)|$$

$$L_2 \text{ (Euclidean) distance: } s(f_1, f_2) = -\|f_1 - f_2\|_2 = -\sqrt{\left(\sum_i (f_1^i - f_2^i)^2\right)}$$

$$\text{Cosine similarity: } s(f_1, f_2) = \frac{f_1^T f_2}{\|f_1\|_2 \|f_2\|_2}$$

Intuitively, since the L2 distance squares the error (increasing it by a lot along dimensions with error  $> 1$ ), the model will see a much larger error than the L1 distance. The model becomes more sensitive to examples with large errors, and adjusts the parameters to minimize this error. If this example is an outlier, the model will be adjusted to minimize this single outlier case, at the expense of many other common examples, since the errors of these common examples are small compared to that single outlier case. On the other hand, cosine similarity measures only the angle between two vectors and is invariant to their magnitude (which is normalized in the denominator). Therefore, it is the least impacted to outliers. The ranking of functions is:

Cosine similarity < L1 distance < L2 distance.

- b. The InfoNCE loss for 1 reference ( $x$ ), 1 positive ( $x^+$ ) and N-1 negative ( $x_j^-$ ) examples is given as follows:

$$\mathcal{L} = -E_X \left[ \log \frac{\exp(s(f(x), f(x^+)))}{\exp(s(f(x), f(x^+))) + \sum_{j=1}^{N-1} \exp(s(f(x), f(x_j^-)))} \right]$$

Consider an alternate loss function where there are no negative examples, such that the denominator is removed from the InfoNCE loss as follows:

$$\mathcal{L} = -E_X[s(f(x), f(x^+))]$$

What is the problem with this loss function in practice, which leads to the need for negative examples during training?

**Solution:**

There exists a trivial solution to minimize this alternate loss function  $s(f(x), f(x^+))$ , which is to simply output  $f(x) = 0$  for all  $x$ . If  $f(x)$  is always zero, the InfoNCE loss becomes  $\log(N)$ , a relatively high value. However, if the second term in the denominator of the InfoNCE loss becomes zero, the total loss also becomes zero. In order to do this,  $s(f(x), f(x_j^-))$  must become a large negative value. Therefore, the InfoNCE loss avoids the trivial solution by encouraging  $f(x)$  and  $f(x_j^-)$  to be different from each other.

## COORDINATE-BASED NETWORKS

**Coordinate-based networks** are a type of neural network architecture used in the field of computer vision. They use a set of 2D or 3D coordinates as input to represent an object or scene, and the network then processes this input to produce output such as object detection, segmentation, or pose estimation. The coordinate-based approach allows the network to learn the relationships between the object parts and their relative positions, which is particularly useful for tasks where the spatial relationships between objects is important. Examples of coordinate-based networks include **Convolutional Neural Networks (CNNs)**, **Graph Convolutional Networks (GCNs)**, and **PointNet**.

## Implicit Neural Representations

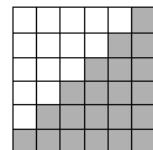
**Implicit neural representation** refers to the internal representations learned by neural networks that are not explicitly defined, but rather learned through training. In computer vision, implicit neural representations are often used in unsupervised or self-supervised learning to discover and extract useful features from image data.

Examples of algorithms that use implicit neural representation in computer vision include:

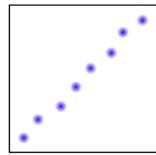
- **Autoencoders**: Unsupervised learning algorithms that learn to reconstruct an input image into a compact representation, capturing meaningful features in the process.
- **Generative Adversarial Networks (GANs)**: Generative models that use two neural networks - a generator and a discriminator - to learn an implicit representation of the data distribution.
- **Variational Autoencoders (VAEs)**: A type of autoencoder that learns a probabilistic encoding of the data, allowing for efficient sampling and generation of new samples.
- **Self-supervised learning**: A type of unsupervised learning that uses auxiliary tasks, such as predicting missing parts of an image, to learn useful representations of the data.

There are few ways to represent a 3D scene, using:

- **Voxels (Volumetric pixels)** are a 3D equivalent of a pixel.



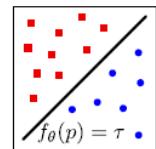
- **Points Cloud** is a discretization of the surface into 3D points. It is not a volumetric representation but it directly represents the surface.



- **Mesh** is a way to discretize the 3D space into vertices and faces to represent the object's surface.



- With **Implicit Representation** the goal is to go away from these discretized representations and instead represent the surface with an implicit function (for example, a decision boundary of a non-linear classifier, that classifies point as in or out of object).



## Differentiable Volumetric Rendering

**Differentiable volumetric rendering** refers to the ability to perform differentiable 3D rendering, where the renderer's output can be treated as a differentiable function and be optimized using gradient-based methods. This allows for the integration of rendering and optimization in a single end-to-end framework, enabling the learning of complex 3D representations and shapes from data.

In computer vision, differentiable volumetric rendering is used for tasks such as:

- **3D Object Generation and Reconstruction:** Generating and reconstructing 3D objects from 2D images or point clouds.
- **Pose Estimation:** Estimating the pose (position and orientation) of 3D objects in an image.

- **3D Scene Understanding:** Understanding and segmenting 3D scenes and objects in images or videos.

Examples of algorithms that use differentiable volumetric rendering in computer vision include:

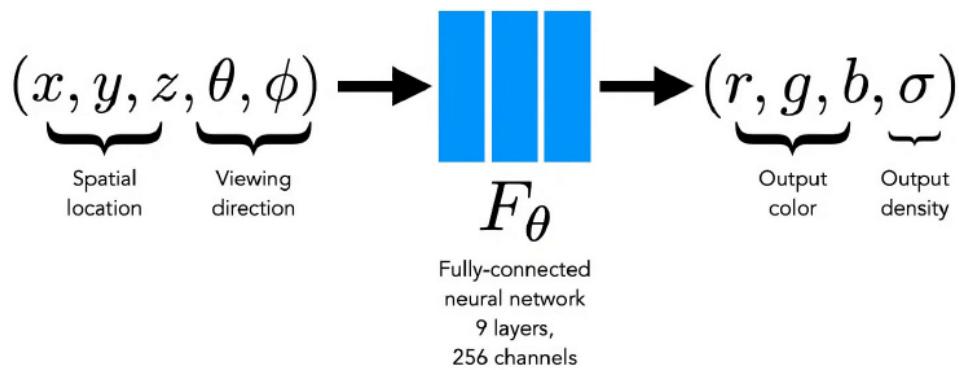
- **Neural 3D Mesh Renderers:** Neural networks that learn to render 3D objects and scenes, allowing for the optimization of both object shapes and camera parameters.
- **Differentiable Ray Tracing:** Differentiable renderers that perform 3D rendering by tracing rays through the scene and using gradients to optimize the scene and object parameters.
- **Implicit Functions:** Neural networks that learn an implicit representation of 3D objects, allowing for the efficient evaluation of the object's shape and optimization of its parameters.

## NeRF

### RAY TRACING (RTX)

**Ray tracing** is a computer graphics technique for rendering images by tracing the path of light ( $\text{ray } r(t) = o + td$ ) as pixels in an image plane and simulating the effects of its encounters with virtual objects. It provides a more physically accurate representation of light and shadows than traditional rasterization methods, producing high-quality images with reflections, refractions, and other optical effects.

**Neural Radiance Fields (NeRF)** is a popular implicit representation method for synthesizing novel views of sparsely sampled images of a scene. Unlike other methods that focus on accurate 3D reconstruction, NeRF focuses on rendering novel viewpoints as realistically as possible. It works for general scenes and does not require pixel-aligned masks of the object, making it more general. NeRF uses a **ReLU MLP** to map location and view direction to color and density, allowing it to model solid or transparent 3D points, view-dependent effects like reflections, and non-Lambertian real-world objects.



The radiance field is estimated along the ray and is used to obtain the pixel color via **alpha composition**. The training is done by shooting rays, rendering the rays to pixels, and minimizing the reconstruction error via backpropagation.

Rendering model for ray  $r(t) = o + td$ :

$$C \approx \sum_{i=1}^N T_i \alpha_i c_i$$

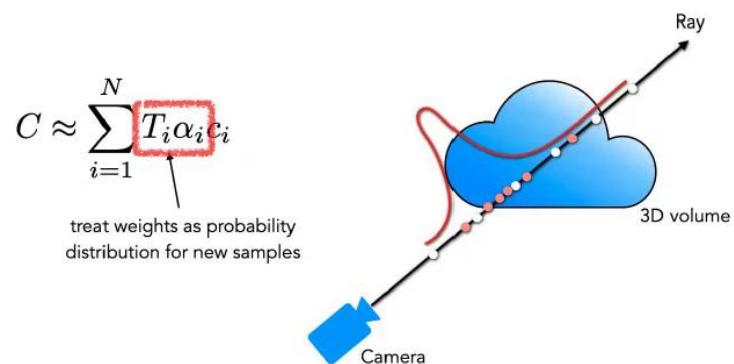
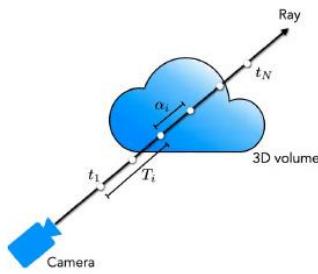
weights      colors

How much light is blocked earlier along ray:

$$T_i = \prod_{j=1}^{i-1} (1 - \alpha_j)$$

How much light is contributed by ray segment  $i$ :

$$\alpha_i = 1 - e^{-\sigma_i \delta t_i}$$



However, NeRF has some drawbacks, such as the requirement for **many different views** of a single scene and **slow sampling**. The two-pass sampling operation helps to mitigate this, by allocating samples more sparsely and densely as NeRF becomes more certain of the surface location.

NeRF can model view-dependent effects like reflections for non-Lambertian materials. This is achieved by providing the view direction as input to the radiance field. The view direction is later conditioned in the MLP to avoid entangling it with the geometry. NeRF also uses **positional encoding** for the input point  $x$  and direction  $d$  to improve its performance. The authors of the NeRF paper showed that a standard fully-connected network with ReLU activation produced over-smoothed results when memorizing a simple image in RGB space. To solve this problem, the authors used positional encoding with **random Fourier features** of varying frequencies instead of passing the position directly into the network. This allowed the network to learn high-frequency functions in low-dimensional domains and produced much sharper and more detailed results with faster convergence.

# IMAGE ENHANCEMENT

## Generative Models

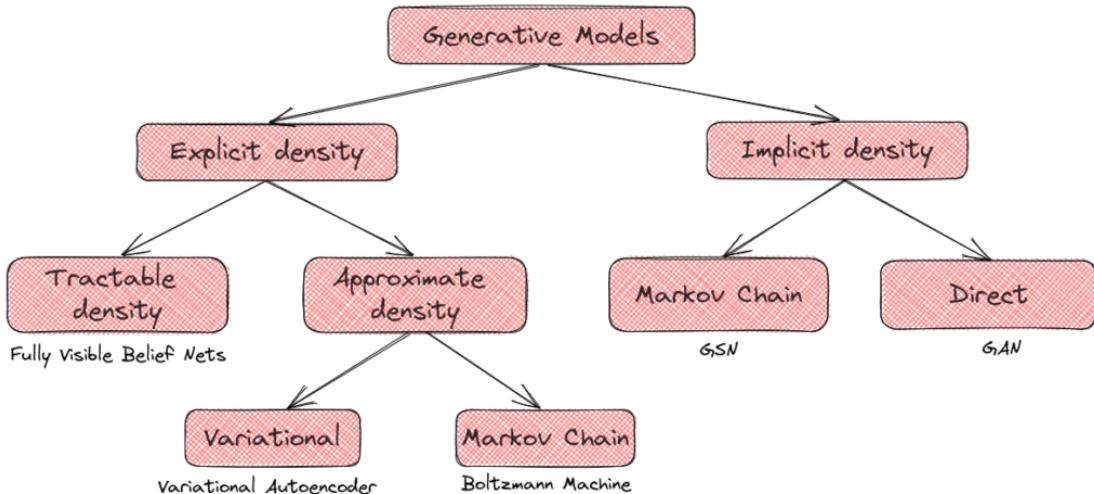
In Machine Learning, there are two major types of learning:

- **Supervised Learning** where we are **given the independent variables  $x$**  and the **corresponding label  $y$**  and our goal is to **learn a mapping function  $f: x \rightarrow y$**  that **minimizes a predefined loss function**. In these tasks, we train discriminative models that aim to learn the conditional probability  $p(y|x)$ . Examples of supervised learning tasks include classification, regression, etc.
- **Unsupervised Learning** where we are **given only the independent variables  $x$**  and our goal is to **learn some underlying patterns of the data**. In these tasks, we train generative models that aim to capture the probability  $p(x)$ . Examples of unsupervised learning tasks include clustering, dimensionality reduction, etc.

Generally, a **generative model tries to learn the underlying distribution of the data**. Then, the model is able to predict how likely a given sample is and to generate some new samples using the learned data distribution.

There are two types of generative models:

- On the one hand, we have the **explicit density models** that assume a prior distribution of the data. Here, **we define an explicit density function**, and then we try to **maximize the likelihood of this function on our data**. If we can define this function in a parametric form, we talk about a **tractable density function**. However, in many cases like images, it is impossible to design a parametric function that captures all the data distribution, and we have to use an approximation of the density function.
- On the other hand, there are the **implicit density models**. These models **define a stochastic procedure that directly generates data**. GANs fall into this category:

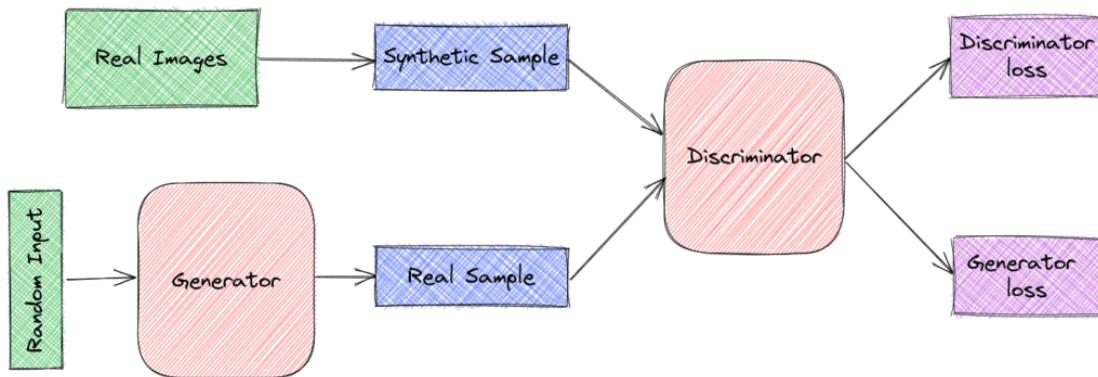


# Generative Adversarial Networks

## ARCHITECTURE

Let's start with the basic architecture of a GAN that consists of two networks:

- First, there is the **Generator** that takes as input a fixed-length random vector  $z$  and learns a mapping  $G(z)$  to produce samples that **mimic the distribution of the original dataset**.
- Then, we have the **Discriminator** that takes as input a sample  $x$  that comes either from the original dataset or from the output distribution of the **Generator**. It outputs a single scalar that **represents the probability that  $x$  came from the original dataset**.



Both  $G$  and  $D$  are differentiable functions that are represented by neural networks.

## LOSS FUNCTIONS

We can think of  $G$  as a team of counterfeiters that produce fake currency while we can compare  $D$  to the police that tries to detect the counterfeit currency. The goal of  $G$  is to deceive  $D$  and use the fake currency without getting caught. Both parties try to improve their methods until, at some point, the fake currency cannot be distinguished from the genuine one.

More formally,  $D$  and  $G$  play a two-player minimax game with the following objective function:

$$\min_G \max_D \log(D(x)) + \log(1 - D(G(z)))$$

where  $x$  comes from the **original dataset** and  $z$  is the **random vector**.

We observe that the objective function is defined using both models' parameters. The goal of  $G$  is to **minimize the term  $\log(1 - D(G(z)))$**  in order to **fool the discriminator** in classifying the fake samples as real ones.

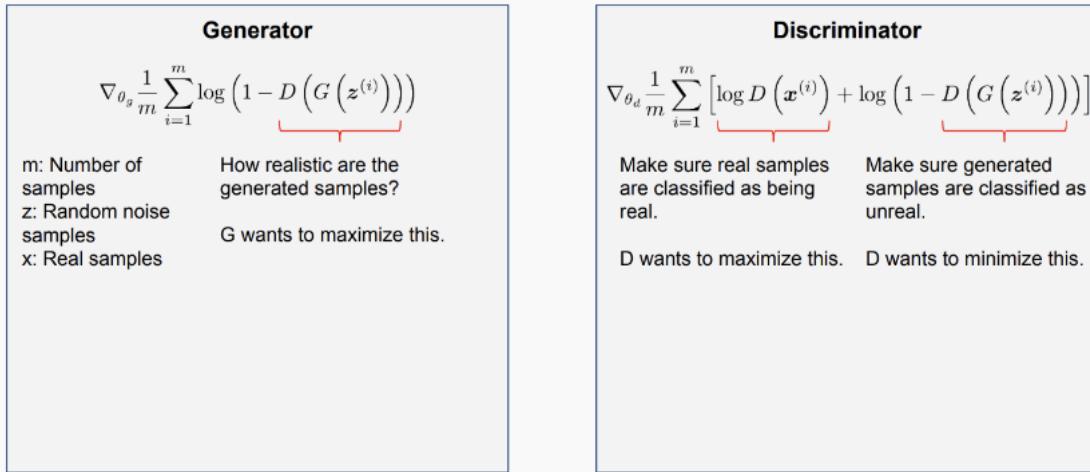
In parallel, the goal of  $D$  is to **maximize  $\log(D(x)) + \log(1 - D(G(z)))$**  that corresponds to the **probability of assigning the correct label** to both the real samples and the samples from the generator.

**It is important to note here that this is not a usual optimization problem since each model's objective function depends on the other model's parameters, and each model controls only its**

**own parameters.** That's why we talk about a game and not an optimization problem. While the solution to an optimization problem is a local or global minimum, the solution here is a [Nash equilibrium](#). Where:

$$P_{data}(x) = p_{gen}(x) \quad \forall x$$

$$D(x) = \frac{1}{2} \quad (\text{Random Classifier} \rightarrow \text{Coin Flip})$$



## TRAINING

**Simultaneous SGD is used for training a GAN.** In each step, we sample two batches of:

- $x$  samples from the original dataset
- $z$  vectors from the prior random distribution

Then, we pass them through the respective models as in the previous figure. Finally, **we apply two gradient steps simultaneously**: one that **updates the parameters of  $G$**  in respect to its objective function and one that **updates the parameters of  $D$** .

**Algorithm 1** Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator,  $k$ , is a hyperparameter. We used  $k = 1$ , the least expensive option, in our experiments.

```

for number of training iterations do
  for  $k$  steps do
    • Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_g(z)$ .
    • Sample minibatch of  $m$  examples  $\{x^{(1)}, \dots, x^{(m)}\}$  from data generating distribution  $p_{data}(x)$ .
    • Update the discriminator by ascending its stochastic gradient:
      
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D \left( x^{(i)} \right) + \log \left( 1 - D \left( G \left( z^{(i)} \right) \right) \right) \right].$$

  end for
  • Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_g(z)$ .
  • Update the generator by descending its stochastic gradient:
    
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log \left( 1 - D \left( G \left( z^{(i)} \right) \right) \right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

```

*Tips for Training GANs*

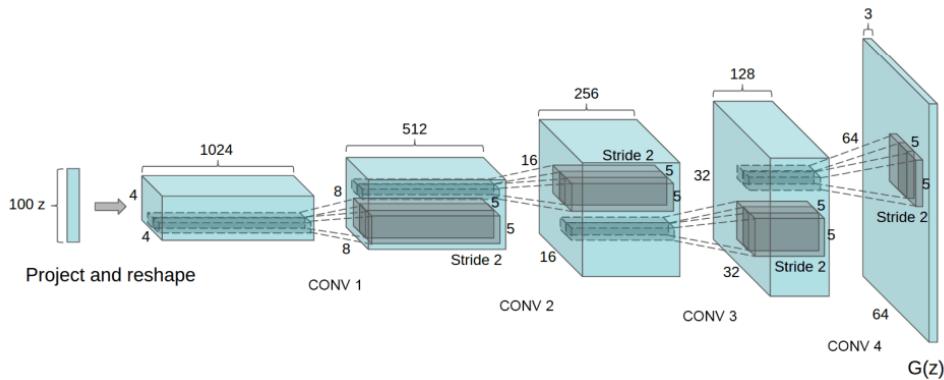
- Normalize the inputs - usually between  $[-1,1]$ . Use TanH for the Generator output.
- Use the modified loss function to avoid the vanishing gradients.
- Use a spherical Z - sample from a Gaussian distribution instead of uniform distribution.
- BatchNorm (when batchnorm is not an option use instance normalization).
- Avoid Sparse Gradients: ReLU, MaxPool - the stability of the GAN game suffers if you have sparse gradients.
  - LeakyReLU is good (in both G and D)
  - For Downsampling, use: Average Pooling, Conv2d + stride
  - For Upsampling, use: PixelShuffle, ConvTranspose2d + stride
- Use Soft and Noisy Labels
  - Label Smoothing, i.e. if you have two target labels: Real=1 and Fake=0, then for each incoming sample, if it is real, then replace the label with a random number between 0.7 and 1.2, and if it is a fake sample, replace it with 0.0 and 0.3.
  - Make the labels the noisy for the discriminator: occasionally flip the labels when training the discriminator
- Track failures early:
  - D loss goes to 0 -- failure mode.
  - Check norms of gradients: if they are over 100 things are not good...
  - When things are working, D loss has low variance and goes down over time vs. having huge variance and spiking.
- Don't balance loss via statistics (unless you have a good reason to)
  - For example, don't do that:

```
while lossD > A: train D or while lossG > B: train G
```

## DCGAN

One of the most well-known GAN architectures for images is the **Deep Convolutional GAN (DCGAN)** which has the following characteristics. **Batch normalization** is applied in all the layers of the **Generator** and the **Discriminator**. Of course, **their output layers are not normalized** in order for them to learn the real mean and scale of the image distribution.

Then the **Generator** uses the convolutional architecture:



During training, **Adam optimizer** is used instead of SGD, which then **uses ReLU activation in the Generator** and **Leaky ReLU activation in the Discriminator**.

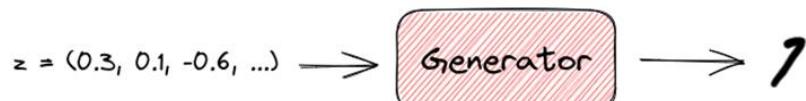
## EXAMPLE

Now let's describe how a GAN learns to generate images of the digit "7".

First, the Generator samples a vector  $z$  from some simple prior distribution and outputs an image  $G(z)$ . **Since the parameters of the model are randomly initialized, the output image is not even close to the digit "7":**



During training, the Generator learns to produce images closer and closer to the original distribution (that depicts the digit "7") in order to fool the Discriminator. So, at some point  $G$  outputs images more similar to the digit "7":



In the end, **the image distribution of the output of the generator and the original distribution is very close to each other, and synthetic images depicting the digit "7" are generated:**



## Applications

Now let's talk about some of the most useful applications of GANs.

### DATA AUGMENTATION

A very important application of generative models is **data augmentation**. In cases when it is difficult or expensive to annotate a large amount of training data, **we can use GANs to generate synthetic data and increase the size of the dataset**.

For example, **StyleGAN** is a generative model proposed by Nvidia that is able to **generate very realistic images of human faces that don't exist**. In the images below, we can see some synthetic faces generated by StyleGAN in different resolutions. I am pretty sure that you can't see any difference between these synthetic facial images and some real facial images. The fact that these people don't actually exist is impressive:



**StyleGAN can also control the style of the generated faces.** Some parts of the network control high-level styles like hairstyle and head pose, while others control facial expression and more fine details. So, the model enables us to modify the style of a person using the style of another person. In the images below, the **style of source A and the identity of source B are combined** to generate a synthetic facial image and the results are very realistic:



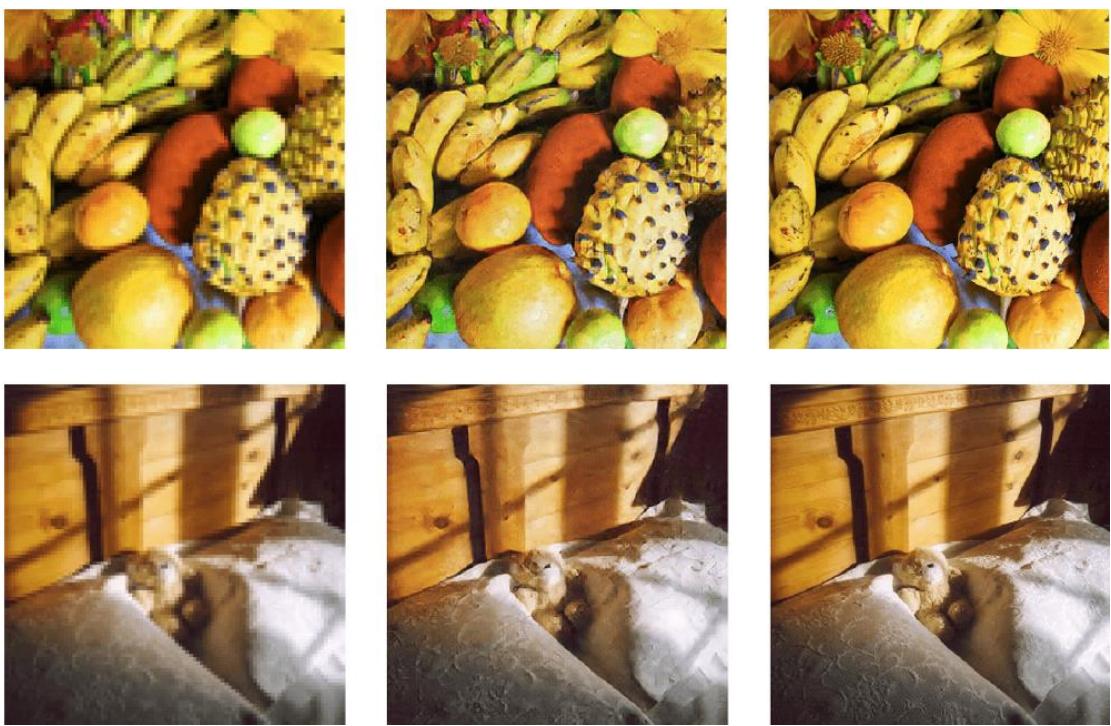
## SUPER RESOLUTION

Another domain where generative models have found many applications is **super-resolution** where our goal is to **enhance the resolution of an input image**. Specifically, we take as input a low-resolution image (like  $90 \times 90$ ) and we want to increase its resolution (to  $360 \times 360$  and even more) and keep its quality as high as possible. Super resolution is a very challenging task with a wide range of applications such as aerial or medical image analysis, video enhancement, surveillance, etc.

### SRGAN

[SRGAN](#) is a generative model that can successfully recover photo-realistic high-resolution images from low-resolution images. The model comprises a deep network in combination with an adversary network like in most GAN architectures.

In the images below, we can see some exciting results of SRGAN. On the left, there is the original low-resolution image. In the middle, there is the generated high-resolution image by SRGAN and on the right, there is the original high-resolution image:



We observe that the generated images are very similar to the original high-resolution ones. The model managed to increase the resolution of the input image without reducing the final quality.

## INPAINTING

In **image inpainting**, our task is to **reconstruct the missing regions in an image**. In particular, we want to fill the missing pixels of the input image in a way that the new image is still realistic and the context is consistent. The applications of this task are numerous like image rendering, editing, or unwanted object removal.

**Deepfill** is **open-source framework for the image inpainting task** that uses a generative model-based approach. Its novelty lies in the **Contextual Attention layer** which allows the generator to make **use of the information given by distant spatial locations for the reconstruction of local missing pixels**.

Below, we can see the effect of Deepfill in 3 different types of images: natural scene, face, and texture:

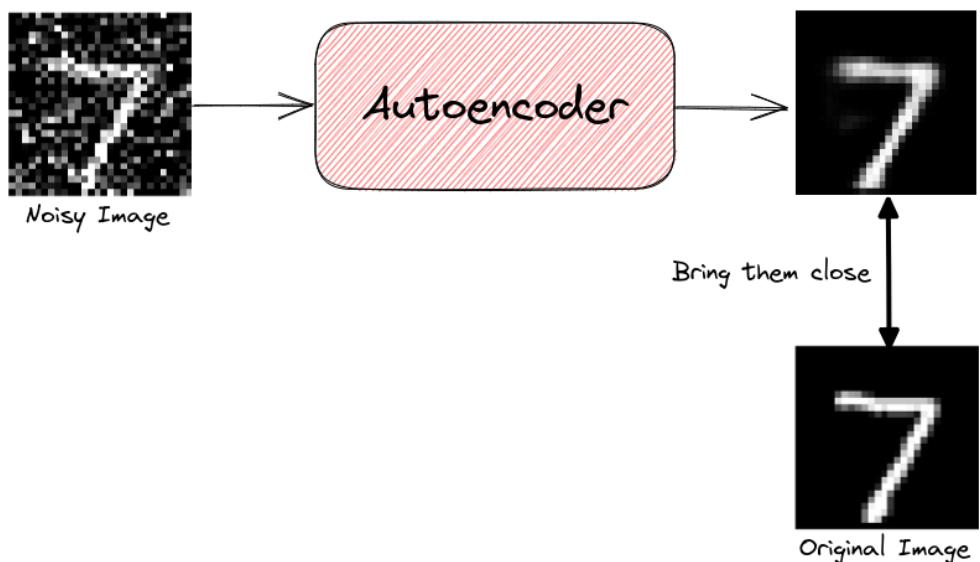


The model is able to fill the missing pixels very naturally keeping the context of the image consistent.

## DENOISING

Nowadays, thanks to modern digital cameras we are able to take high-quality photos. However, there are still cases where an image contains a lot of noise and its quality is low. **Removing the noise from an image without losing image features** is a very crucial task and researchers have been working on denoising methods for many years.

A very popular generative model for image denoising is [Autoencoder](#) that is **trained to reconstruct its input image after removing the noise**. During training, the network is given the original image and its noisy version. Then, the network tries to reconstruct its output to be as close as possible to the original image. As a result, the model learns to denoise images:



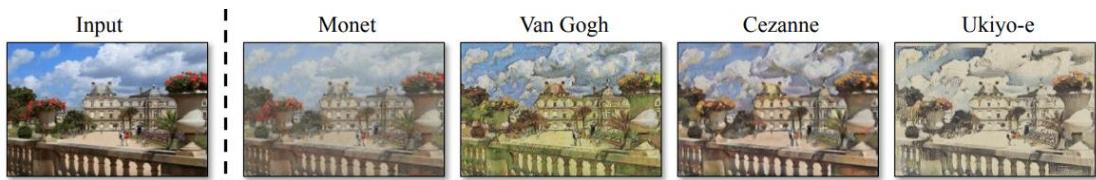
## TRANSLATION

Last but not least, generative models are also used in **image translation** where our goal is to **learn the mapping between two image domains**. Then, the model is able to **generate a synthetic version of the input image with a specific modification** like translating a winter landscape to summer.

CycleGAN is a very famous **GAN-based model for image translation**. The model is trained in an unsupervised way using a dataset of images from the source and the target domain. The applications that derive from this method are above your imagination!

## Collection Style Transfer

Here, the model takes as input a **random landscape and transforms it into a painting of a famous painter** like Monet, Van Gogh, Cezanne, and Ukiyo-e:

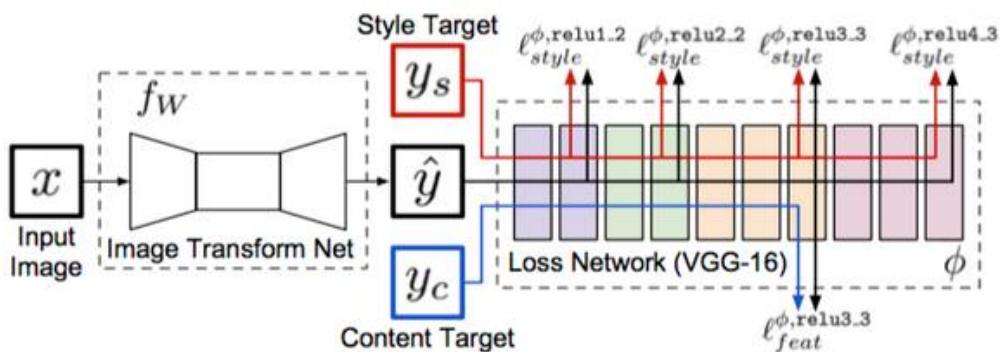


### Perceptual loss

**Perceptual loss** functions are used when **comparing two different images that look similar**, like the same photo but shifted by one pixel. The function is used to **compare high level differences**, like content and style discrepancies, between images. A perceptual loss function is very similar to the per-pixel loss function, as both are used for training feed-forward neural networks for image transformation tasks. The perceptual loss function is a more commonly used component as it often provides more accurate results regarding **style transfer**.

### How does a Perceptual Loss Function work?

In short, the perceptual loss function works by **summing all the squared errors between all the pixels and taking the mean**. This is in contrast to a per-pixel loss function which sums all the absolute errors between pixels. [Johnson](#) argues that perceptual loss functions are not only more accurate in generating high quality images, but also do so as much as three times faster, when optimized. The neural network model is trained on images where the perceptual loss function is optimized based upon high level features extracted from already trained networks.



The image above represents the neural network that is trained to transform input images into output images. A **pre-trained loss network** used for image classification helps inform the loss functions. The pre-trained network helps to define the perceptual loss functions needed to measure the perceptual differences of the content and style between the images.

### *Object Transfiguration*

Another exciting application of StyleGAN is object transfiguration where the model **translates one object class to another** like translating a horse to a zebra, a winter landscape to a summer one, and apples to oranges:



### *Image Colorization*

We can use CycleGAN for **automatic image colorization** which is very useful in areas like restoration of aged or degraded images. Below, CycleGAN converts a grayscale image of a flower to its colorful RGB form:



## Challenges

Although GANs have already succeeded in many areas, there are yet many challenges that we have to face when training a GAN for unsupervised learning.

### NON-CONVERGENCE

As we mentioned earlier, training a GAN is **not an ordinary optimization problem but a minimax game**. So, **achieving convergence to a point that optimizes the objectives of both the generator and the discriminator is challenging since we have to learn a point of equilibrium**. Theoretically, we know that simultaneous SGD converges if the updates are made in [function space](#). When using neural networks, this hypothesis does not apply, and **convergence is not theoretically guaranteed**. Also, many times we observe cases where the Generator undoes the progress of the Discriminator without arriving anywhere useful and vice versa.

## EVALUATION

One important aspect of every learning task is **evaluation**. We can qualitatively evaluate a GAN quite easily by inspecting the synthetic samples that the generator produces. However, we need quantitative metrics in order to robustly evaluate any model. Unfortunately, it is not clear **how to quantitatively evaluate generative models**. Sometimes, **models that obtain good likelihood generate unrealistic samples**, and other models that **generate realistic samples present poor likelihood**.

## DISCRETE OUTPUTS

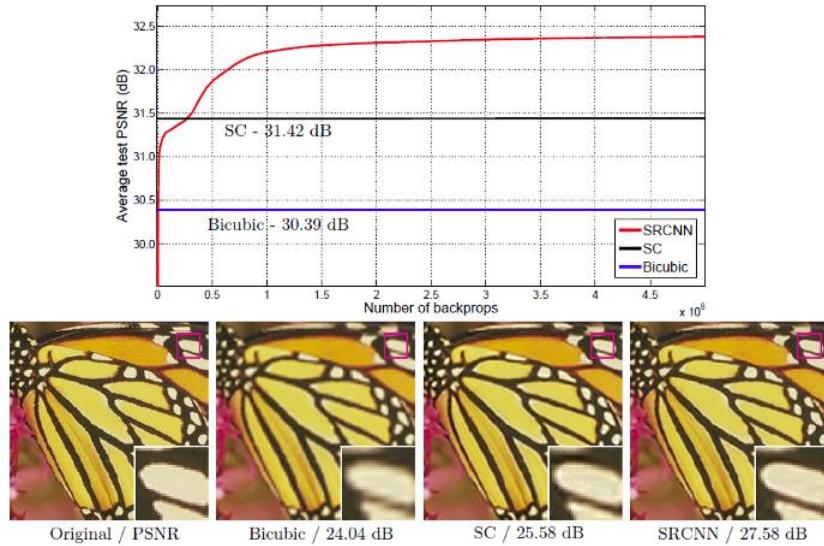
To train a GAN, the architecture of the generator should be differentiable. However, **if we want the generator to produce discrete data, the respective function won't be differentiable**. Although there are many proposed solutions to this restriction, there is no optimal universal solution. Dealing with this problem will help us use GANs for domains like NLP.

## USING THE CODE

The Generator takes as input a random vector  $z$  and generates a sample  $x$ . Therefore, the vector  $z$  can be considered as a feature representation of the sample  $x$  and can be used in a variety of other tasks. However, it is very difficult to obtain  $z$  given a sample  $x$  since **we want to move from a high-dimensional space to a low-dimensional one**.

## Super-Resolution Convolutional Neural Network (SRCNN)

In SRCNN, the convolutional neural network is used for **single image super resolution (SR)** which is a classical problem in computer vision. In brief, with better SR approach, we can get a better quality of a larger image even we only get a small image originally.



We can see from the above figure that, with SRCNN, PSNR of 27.58 dB is obtained which is much better than the classical non-learning based Bicubic and sparse coding (SC) which was and still is also a very hot research topic.

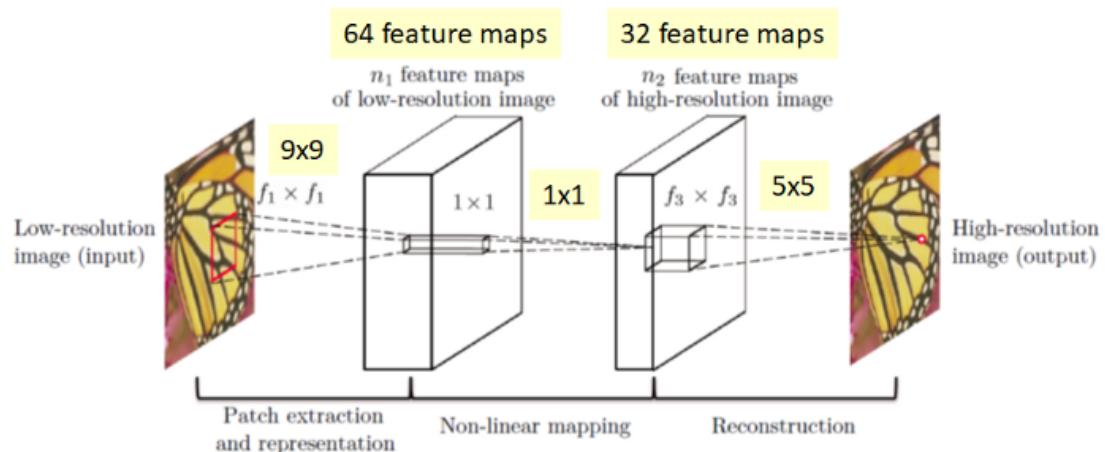
SRCNN is published in [2014 ECCV](#) and [2016 TPAMI](#) papers with both about 1000 citations.

### SRCNN NETWORK

In SRCNN, actually the network is not deep. There are only 3 parts:

- **Patch extraction and representation**
- **Non-linear mapping**
- **Reconstruction**

as shown in the figure below:



### *Patch Extraction and Representation*

It is important to know that the **low-resolution input is first upscaled to the desired size using bicubic interpolation** before inputting to SRCNN network. Thus:

- $x$ : **Ground truth high-resolution image**
- $y$ : **Bicubic upsampled version of low-resolution image**

And the first layer perform a standard convolution with ReLU to get  $F_1(y)$ :

$$F_1(y) = \max(0, W_1 * y + B_1)$$

Size of  $W_1$ :  $c \times f_1 \times f_1 \times n_1$

Size of  $B_1$ :  $n_1$

Where  $c$  is **number of channels of the image**,  $f_1$  is the **filter size**, and  $n_1$  is the **number of filters**.  $B_1$  is the  **$n_1$ -dimensional bias vector** which is just used for increasing the degree of freedom by 1.

In this case:  $c = 1, f_1 = 9, n_1 = 64$ .

### *Non-Linear Mapping*

After that, a **non-linear mapping** is performed:

$$F_2(y) = \max(0, W_2 * F_1(y) + B_2)$$

Size of  $W_2$ :  $n_1 \times 1 \times 1 \times n_2$

Size of  $B_2$ :  $n_2$

It is a mapping of  $n_1$ -dimensional vector to  $n_2$ -dimensional vector. When  $n_1 > n_2$ , we can imagine something like PCA stuffs but in a non-linear way.

In this case,  $n_2 = 32$ .

This  $1 \times 1$  actually is a  $1 \times 1$  convolution suggested in [Network In Network \(NIN\)](#) as well. In NIN,  $1 \times 1$  convolution is suggested to **introduce more non-linearity to improve the accuracy**. It is also suggested in [GoogLeNet](#) for **reducing the number of connections**.

Here, it is used for **mapping low-resolution vector to high-resolution vector**.

### *Reconstruction*

After mapping, we need to **reconstruct the image**. Hence, we do convolution again:

$$F_3(y) = W_3 * F_2(y) + B_3$$

Size of  $W_3$ :  $n_2 \times f_3 \times f_3 \times c$

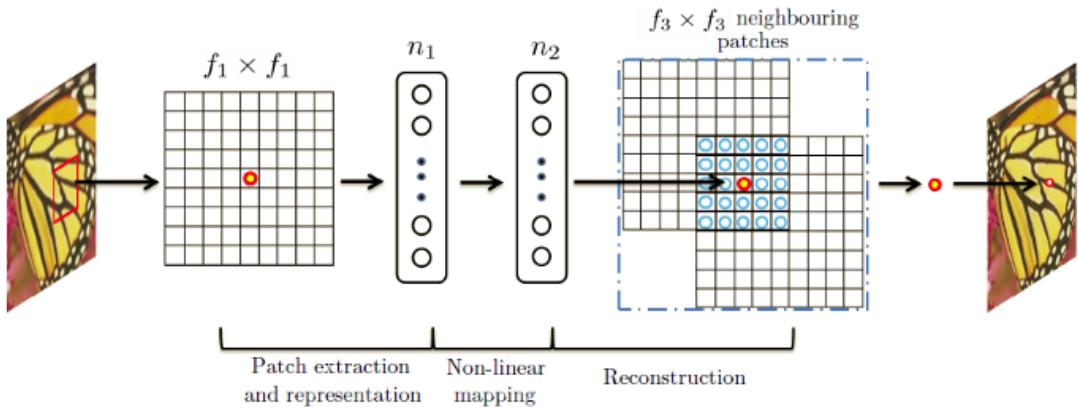
Size of  $B_3$ :  $c$

## LOSS FUNCTION

$$L(\theta) = \frac{1}{n} \sum_{i=1}^n \|F(y_i; \theta) - x_i\|^2$$

For **super resolution**, the **loss function  $L$**  is the **average of mean square error (MSE)** for the training samples ( $n$ ), which is a kind of standard loss function.

## RELATION WITH SPARSE CODING



For **Sparse Coding (SC)**, in the view of convolution, the input image is convolution by  $f_1$  and project to onto a  $n_1$ -dimensional dictionary.  $n_1 = n_2$  usually is the case of SC. Then mapping of  $n_1$  to  $n_2$  is done with the same dimensionality without reduction. It is just like a mapping of low-resolution vector to high-resolution vector. Then each patch is reconstructed by  $f_3$ . And **overlapping patches are averaged instead of adding together with different weights by convolution.**

## Autoencoders

**Information theory** is a scientific field of great interest as it is expected to provide answers for today's data storage and data analysis problems. It is this field that provides the theoretical base for **data compression**, after all. There are purely statistical approaches but **machine learning** offers flexible neural network structures that can compress data for a variety of applications.

### FUNCTION

There are many ways to **compress data** and purely statistical methods such as **principal component analysis (PCA)** can help **identify the key features accountable for the variability in the data** and **use these to represent the information using fewer bits**. This type of compression can be called "**Dimensionality Reduction**". However, this PCA solution can only offer an **encoding with linearly uncorrelated features**.

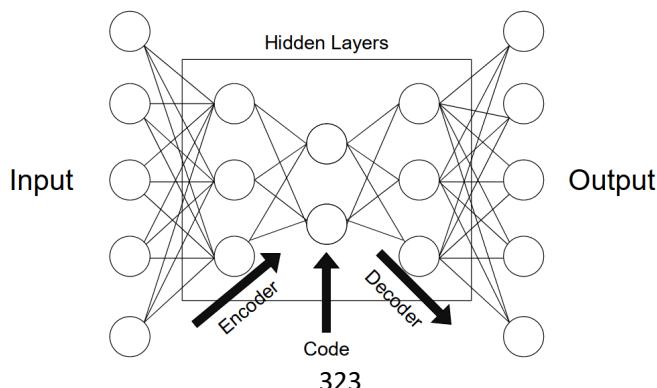
The machine learning alternative solution proposes neural networks that are structured as autoencoder models. **Autoencoders can learn richer, non-linear encoding features**. These features can correlate with each other and are therefore **not necessarily orthogonal** to one another. Using these functions we can represent complex data in a **latent space**.

For example, when discussing Convolutional Neural Networks (CNNs) for facial recognition, these can be used as autoencoders for image data. They can allow a user to store photos using less space by encoding the image by lowering the quality just a little bit. The **difference or loss in quality between the output and input images** is called **reconstruction loss**.

The main goal for autoencoders is to **represent complex data using as little code as possible** with little to **no reconstruction** or "**compression**" loss. To do so, the autoencoder has to look at the data and construct a function that can transform a particular instance of data into a meaningful code. We can think of this as a remapping of the original data using fewer dimensions. We can also keep in mind that this code has to be interpreted later on by a decoder to access the data.

### GENERAL STRUCTURE

To better understand the functioning of these models, we'll **decompose them into individual components**. We'll describe the structure in the same order that the data travels our neural net, discussing **encoders** first, then the **bottleneck** or "**code**" and then **decoders**. These segments of autoencoders can be seen as different layers of nodes in a model. The image below helps interpret these three different components. The arrow indicates the flow of the data in our model:



### *Encoder*

The **encoder**'s role is to **compress the data into a code**. In neural networks, we can implement this phenomenon by connecting a series of pooling layers, each one reducing the number of dimensions that are present in the data. In doing this, we can **de-noise data by keeping only the parts of it that are relevant enough to encode**.

The layer of the neural network that has the fewest dimensions, generally in the middle of all the layers, is called the **bottleneck**. If we're interested in keeping more information about a specific instance, we'd need a **larger code size**. This "**code size**" hyper-parameter is important as it defines how much data gets to be encoded and simultaneously regulates our model. **A very big code will represent more noise but too little code may not accurately represent the data at hand.**

### *Decoder*

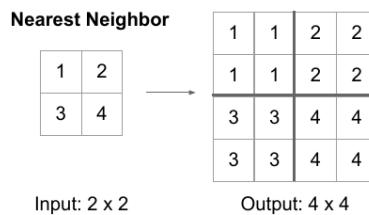
The **decoder** component of the network acts as an **interpreter for the code**. In the case of convolutional neural nets, it can reconstruct an image based on a particular code. We can think of this component as a value extraction, interpretation, or decompression tool. Image segmentation is usually the decoders' job as well. If we keep our last example of classifying dog pictures, we could continue decoding while adding a copy of previous compressed layers to each layer of decoder nodes. In doing so, we should obtain a photograph with a subset of pixels that identify the dog in the picture and segment it from the background and other objects.

**Generative models** such as **Variational Auto-Encoders (VAEs)** can even **use the decoder to render data that doesn't exist**. This can be useful for **data augmentation** purposes. In having diverse data sets from generative models, other models can learn more thoroughly from data.

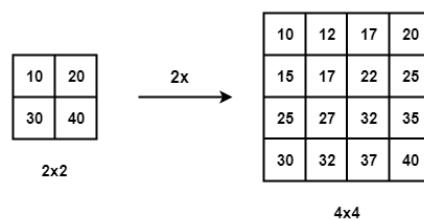
### Upsampling Techniques

The most widely used techniques for upsampling in Encoder-Decoder Networks are:

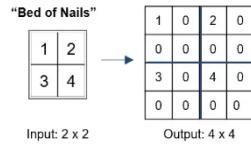
1. **Nearest Neighbors:** In Nearest Neighbors, as the name suggests we take an input pixel value and copy it to the K-Nearest Neighbors where K depends on the expected output.



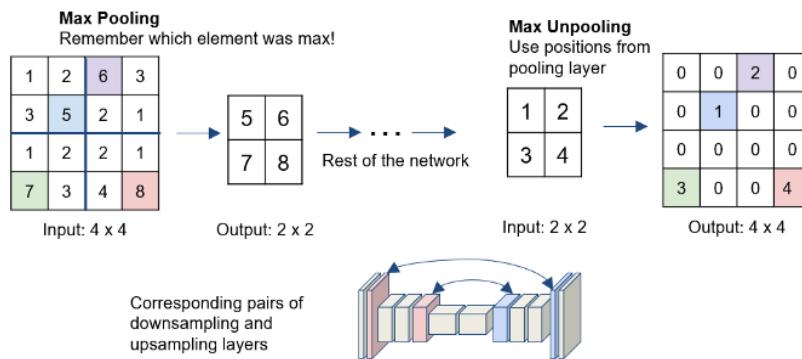
2. **Bi-Linear Interpolation:** In Bi-Linear Interpolation, we take the 4 nearest pixel value of the input pixel and perform a weighted average based on the distance of the four nearest cells smoothing the output.



3. **Bed Of Nails:** In Bed of Nails, we copy the value of the input pixel at the corresponding position in the output image and filling zeros in the remaining positions.



4. **Max-Unpooling:** The Max-Pooling layer in CNN takes the maximum among all the values in the kernel. To perform max-unpooling, first, the index of the maximum value is saved for every max-pooling layer during the encoding step. The saved index is then used during the Decoding step where the input pixel is mapped to the saved index, filling zeros everywhere else.



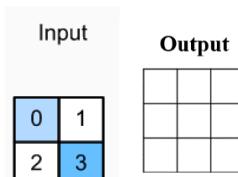
All the above-mentioned techniques are predefined and **do not depend on data**, which makes them task-specific. They do not learn from data and hence are **not a generalized technique**.

#### *Transposed Convolutions (Up-Convolutions)*

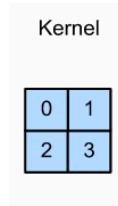
Transposed Convolutions are used to upsample the input feature map to a desired output feature map using some learnable parameters.

The basic operation that goes in a transposed convolution is explained below:

1. Consider a  $2 \times 2$  encoded feature map which needs to be upsampled to a  $3 \times 3$  feature map.



2. We take a kernel of size  $2 \times 2$  with unit stride and zero padding.



3. Now we take the upper left element of the input feature map and multiply it with every element of the kernel as shown in the following figure.

$$\begin{array}{|c|} \hline 0 \\ \hline \end{array} \quad \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} \quad = \quad \begin{array}{|c|c|} \hline 0 & 0 \\ \hline 0 & 0 \\ \hline \end{array}$$

4. Similarly, we do it for all the remaining elements of the input feature map as depicted in the following figure.

$$\begin{array}{|c|c|} \hline \text{Input} & \text{Kernel} \\ \hline \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} & \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} \\ \hline \end{array} = \begin{array}{|c|c|} \hline 0 & 0 \\ \hline 0 & 0 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 0 & 2 \\ \hline 4 & 6 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 0 & 3 \\ \hline 6 & 9 \\ \hline \end{array}$$

5. As you can see, some of the elements of the resulting upsampled feature maps are overlapping. To solve this issue, we simply add the elements of the over-lapping positions.

$$\begin{array}{|c|c|} \hline \text{Input} & \text{Kernel} \\ \hline \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} & \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} \\ \hline \end{array} = \begin{array}{|c|c|} \hline 0 & 0 \\ \hline 0 & 0 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 0 & 2 \\ \hline 4 & 6 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 0 & 3 \\ \hline 6 & 9 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 0 & 0 & 1 \\ \hline 0 & 4 & 6 \\ \hline 4 & 12 & 9 \\ \hline \end{array}$$

6. The resulting output will be the final upsampled feature map having the required spatial dimensions of  $3 \times 3$ .

**Transposed convolution** is also known as **Deconvolution** which is not appropriate as deconvolution implies removing the effect of convolution which we are not aiming to achieve.

It is also known as **upsampled convolution** which is intuitive to the task it is used to perform, i.e. upsample the input feature map.

It is also referred to as **fractionally strided convolution** since stride over the output is equivalent to fractional stride over the input. For instance, a stride of 2 over the output is  $1/2$  stride over the input.

Finally, it is also referred to as **Backward strided convolution** because forward pass in a Transposed Convolution is equivalent to backward pass of a normal convolution.

#### Problems with Transposed Convolution

Transposed convolutions suffer from checkered board effects as shown below:

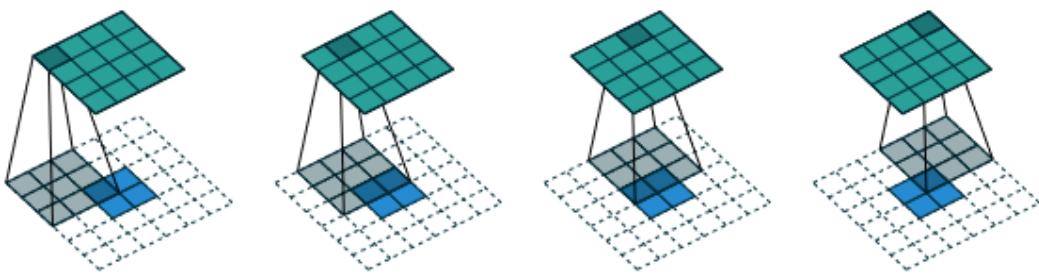


The main cause of this is uneven overlap at some parts of the image causing artifacts. This can be fixed or reduced by using kernel-size divisible by the stride, for e.g taking a kernel size of  $2 \times 2$  or  $4 \times 4$  when having a stride of 2.

Transposed Convolution Arithmetic's

NO ZERO PADDING, UNIT STRIDES, TRANSPOSED

The transpose of convolving a  $3 \times 3$  kernel over a  $4 \times 4$  input using unit strides (i.e.,  $i = 4$ ,  $k = 3$ ,  $s = 1$  and  $p = 0$ ). It is equivalent to convolving a  $3 \times 3$  kernel over a  $2 \times 2$  input padded with a  $2 \times 2$  border of zeros using unit strides (i.e.,  $i' = 2$ ,  $k' = 3$ ,  $s' = 1$  and  $p' = 2$ ).



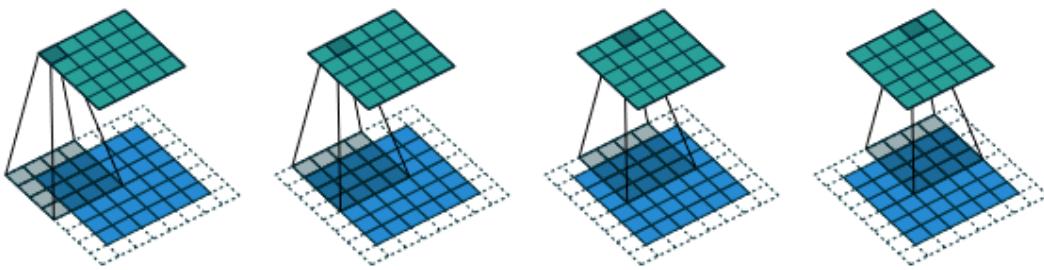
#### *RELATIONSHIP*

A convolution described by  $s = 1$ ,  $p = 0$  and  $k$  has an associated transposed convolution described by  $k' = k$ ,  $s' = s$  and  $p' = k - 1$  and its output size is:

$$o' = i' + (k - 1)$$

ZERO PADDING, UNIT STRIDES, TRANSPOSED

The transpose of convolving a  $4 \times 4$  kernel over a  $5 \times 5$  input padded with a  $2 \times 2$  border of zeros using unit strides (i.e.,  $i = 5$ ,  $k = 4$ ,  $s = 1$  and  $p = 2$ ). It is equivalent to convolving a  $4 \times 4$  kernel over a  $6 \times 6$  input padded with a  $1 \times 1$  border of zeros using unit strides (i.e.,  $i' = 6$ ,  $k' = 4$ ,  $s' = 1$  and  $p' = 1$ ).



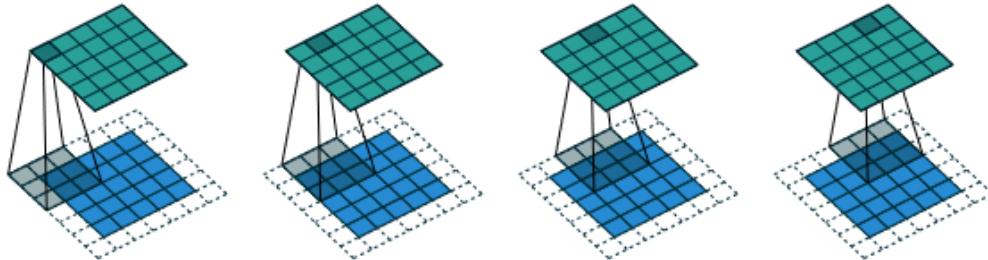
#### *RELATIONSHIP*

A convolution described by  $s = 1$ ,  $k$  and  $p$  has an associated transposed convolution described by  $k' = k$ ,  $s' = s$  and  $p' = k - p - 1$  and its output size is:

$$o' = i' + (k - 1) - 2p$$

## HALF (SAME) PADDING, TRANSPOSED

The transpose of convolving a  $3 \times 3$  kernel over a  $5 \times 5$  input using half padding and unit strides (i.e.,  $i = 5$ ,  $k = 3$ ,  $s = 1$  and  $p = 1$ ). It is equivalent to convolving a  $3 \times 3$  kernel over a  $5 \times 5$  input using half padding and unit strides (i.e.,  $i' = 5$ ,  $k' = 3$ ,  $s' = 1$  and  $p' = 1$ ).



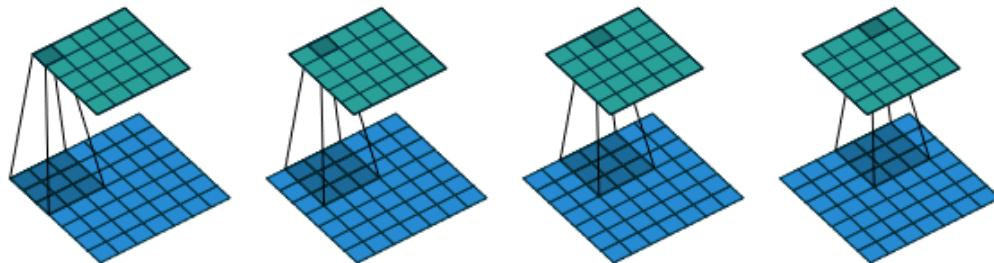
## RELATIONSHIP

A convolution described by  $s = 1$ ,  $k = 2n + 1$  ( $n \in \mathbb{N}$ ) and  $p = \left\lfloor \frac{k}{2} \right\rfloor$  has an associated transposed convolution described by  $k' = k$ ,  $s' = s$  and  $p' = p$  and its output size is:

$$o' = i' + (k - 1) - 2p = i' + 2n - 2n = i'$$

## FULL PADDING, TRANSPOSED

The transpose of convolving a  $3 \times 3$  kernel over a  $5 \times 5$  input using full padding and unit strides (i.e.,  $i = 5$ ,  $k = 3$ ,  $s = 1$  and  $p = 2$ ). It is equivalent to convolving a  $3 \times 3$  kernel over a  $7 \times 7$  input using unit strides (i.e.,  $i' = 7$ ,  $k' = 3$ ,  $s' = 1$  and  $p' = 0$ ).



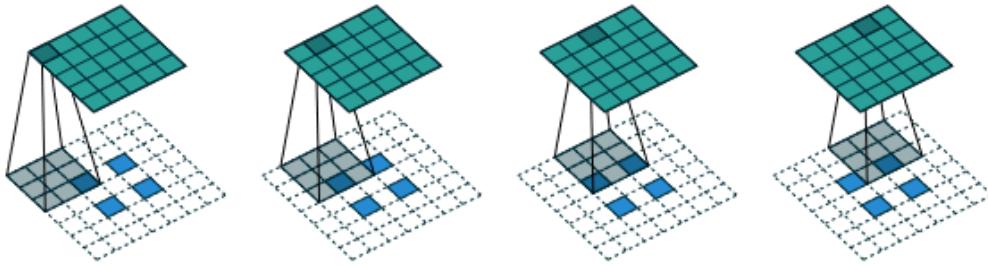
## RELATIONSHIP

A convolution described by  $s = 1$ ,  $k$  and  $p = k - 1$  has an associated transposed convolution described by  $k' = k$ ,  $s' = s$  and  $p' = 0$  and its output size is:

$$o' = i' + (k - 1) - 2p = i' - (k - 1)$$

## NO ZERO PADDING, NON-UNIT STRIDES, TRANSPOSED

The transpose of convolving a  $3 \times 3$  kernel over a  $5 \times 5$  input using  $2 \times 2$  strides (i.e.,  $i = 5$ ,  $k = 3$ ,  $s = 2$  and  $p = 0$ ). It is equivalent to convolving a  $3 \times 3$  kernel over a  $2 \times 2$  input (with 1 zero inserted between inputs) padded with a  $2 \times 2$  border of zeros using unit strides (i.e.,  $i' = 2$ ,  $k' = 3$ ,  $s' = 1$  and  $p' = 2$ ).

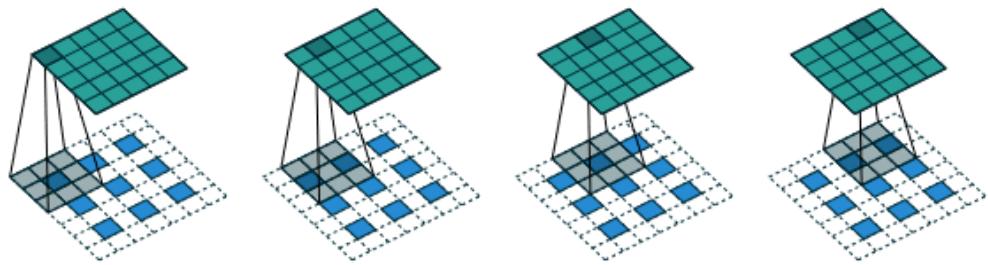
***RELATIONSHIP***

A convolution described by  $p = 0$ ,  $k$  and  $s$  whose input size is such that  $i - k$  is a multiple of  $s$  has an associated transposed convolution described by  $\tilde{i}'$ ,  $k' = k$ ,  $s' = 1$  and  $p' = k - 1$ , where  $\tilde{i}'$  is the size of the stretched input obtained by adding  $s - 1$  zeros between each input unit, and its output size is:

$$o' = s(i' - 1) + k$$

**ZERO PADDING, NON-UNIT STRIDES, TRANSPOSED**

The transpose of convolving a  $3 \times 3$  kernel over a  $5 \times 5$  input padded with a  $1 \times 1$  border of zeros using  $2 \times 2$  strides (i.e.,  $i = 5$ ,  $k = 3$ ,  $s = 2$  and  $p = 1$ ). It is equivalent to convolving a  $3 \times 3$  kernel over a  $3 \times 3$  input (with 1 zero inserted between inputs) padded with a  $1 \times 1$  border of zeros using unit strides (i.e.,  $i' = 3$ ,  $\tilde{i}' = 5$ ,  $k' = 3$ ,  $s' = 1$  and  $p' = 1$ ).

***RELATIONSHIP***

A convolution described by  $k$ ,  $s$  and  $p$  and whose input size  $i$  is such that  $i + 2p - k$  is a multiple of  $s$  has an associated transposed convolution described by  $\tilde{i}'$ ,  $k' = k$ ,  $s' = 1$  and  $p' = k - p - 1$ , where  $\tilde{i}'$  is the size of the stretched input obtained by adding  $s - 1$  zeros between each input unit, and its output size is:

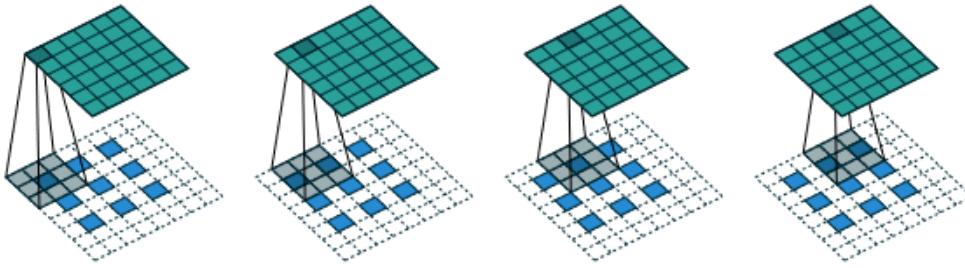
$$o' = s(i' - 1) + k - 2p$$

**MORE GENERAL FORM**

The constraint on the size of the input  $i$  can be relaxed by introducing another parameter  $a \in \{0, \dots, s - 1\}$  that allows to distinguish between the  $s$  different cases that all lead to the same  $i'$ .

The transpose of convolving a  $3 \times 3$  kernel over a  $6 \times 6$  input padded with a  $1 \times 1$  border of zeros using  $2 \times 2$  strides (i.e.,  $i = 6$ ,  $k = 3$ ,  $s = 2$  and  $p = 1$ ). It is equivalent to convolving a  $3 \times 3$  kernel over a  $2 \times 2$  input (with 1 zero inserted between inputs) padded with a  $1 \times 1$

border of zeros (with an additional border of size 1 added to the bottom and right edges) using unit strides (i.e.,  $i' = 3$ ,  $\tilde{i}' = 5$ ,  $a = 1$ ,  $k' = 3$ ,  $s' = 1$  and  $p' = 1$ ).



#### RELATIONSHIP

A convolution described by  $k$ ,  $s$  and  $p$  has an associated transposed convolution described by  $a, \tilde{i}', k' = k, s' = 1$  and  $p' = k - p - 1$ , where  $\tilde{i}'$  is the size of the stretched input obtained by adding  $s - 1$  zeros between each input unit, and  $a = (i + 2p - k) \bmod s$  represents the number of zeros added to the bottom and right edges of the input, and its output size is:

$$o' = s(i' - 1) + a + k - 2p$$

#### Transposed Convolution vs Dilated Convolution

Both **dilated convolution** (atrous convolution) and deconvolution involve convolutional operations, they serve different purposes. Dilated convolution is used for increasing the receptive field and capturing features at multiple scales, while deconvolution is used for upsampling feature maps.

#### Output Size

Upsampled convolution that gets an input size of  $W_{in} \times H_{in}$  is zero padding  $\left(\frac{1}{s}\right) - 1$  pixels between each pixel, hence we get output size of:

$$W_{mid} = (W_{in} - 1) \cdot \frac{1}{S} + 1$$

$$H_{mid} = (H_{in} - 1) \cdot \frac{1}{S} + 1$$

On this size, apply normal convolution, with the size of the kernel ( $K \times K$ ), output padding of  $P$  and **no stride ( $S = 1$ )**, and get the output size of:

$$W_{out} = \frac{W_{mid} - K + 2P}{1} + 1$$

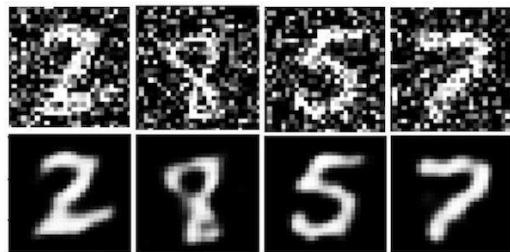
$$H_{out} = \frac{H_{mid} - K + 2P}{1} + 1$$

#### DIFFERENT TYPES

Autoencoders are applied in many different fields across machine learning and computer science. Here are the main types that we can encounter and their respective common applications.

### Denoising Autoencoder

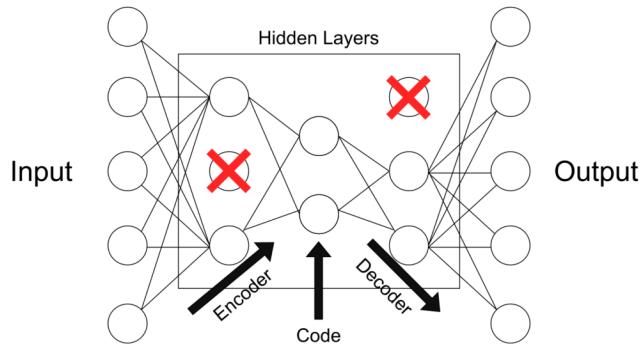
These types of autoencoders are meant to **encode noisy data efficiently to leave random noise out of the code**. In doing so, the output of the autoencoder is meant to be de-noised and therefore different than the input. We can see what an implementation of this would look like using the popular MNIST dataset, as presented in the image below:



These types of autoencoders can be used for feature extraction and data de-noising.

### Sparse Autoencoder

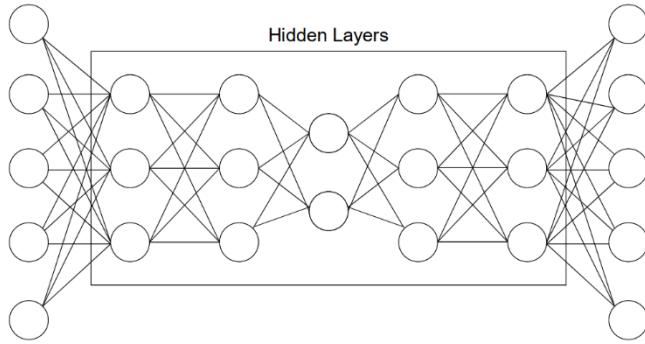
This type of autoencoder explicitly **penalizes the use of hidden node connections**. This **regularizes the model, keeping it from overfitting the data**. This “**sparsity penalty**” is added to the **reconstruction loss** to obtain a global loss function. Alternatively, one could just delete a set number of connections in the hidden layer:



This is more of a regularization method that can be used with a broad array of encoder types. Applications may vary.

### Deep Autoencoder

**Deep autoencoders** are composed of **two symmetrical deep-belief networks**. This structure is similar to the “**general structure**” representation of an autoencoder using nodes and connections found above. These mirrored components can be described as two restricted Boltzmann machines acting as encoders and decoders:



Autoencoders of this type are used for a wide array of purposes such as feature extraction, dimensionality reduction, and data compression.

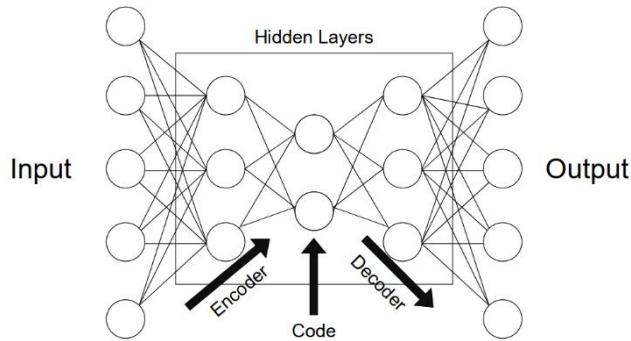
### *Contractive Autoencoder*

**Contractive autoencoders penalize big variations in code when small changes in the input occur.** This means that **similar inputs should have similar codes**. This usually means that the information captured by the autoencoder is meaningful and represents a large variance in the data. To implement this, we can add a penalty loss to the global loss function of the autoencoder.

Data augmentation is a great type of application for these types of autoencoders.

### *Undercomplete Autoencoder*

These autoencoders have **smaller hidden dimensions in comparison to input**. This means that they **excel at capturing only the most important features present in the data**. These types of autoencoders usually do not need regularization as they do not aim to reproduce outputs similar to inputs but rather **rely on the compression stage to capture meaningful features in the data**:



Feature extraction is the main type of application for this type of autoencoders.

### *Variational Autoencoder*

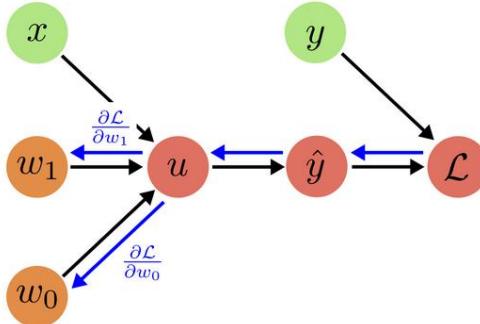
**Variational autoencoders** or **VAEs** assume encode data as distributions as opposed to single points in space. This is a way of having a more regular latent space that we can **better use to generate new data** with. Hence, VAEs are referred to as [generative models](#). The original “[Auto-Encoding Variational Bayes](#)” paper published by Diederik P Kingma and Max Welling details the complete functioning of these particular models.

The process to achieve this **latent space distribution** makes the training process a little different. First, **instead of mapping an instance as a point in space, it is mapped as the center of a normal distribution**. Next, **we take a point from that distribution to decode and compute the reconstruction error that will be backpropagated across the network**.

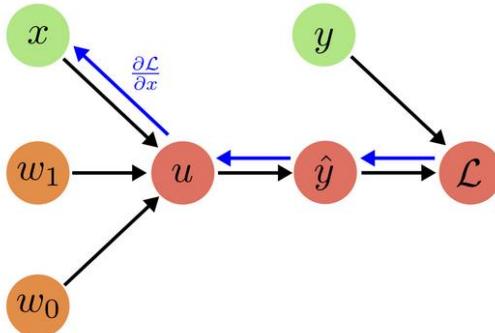
## DIVERSE TOPICS IN COMPUTER VISION

### Input Optimization

Optimization with Neural Networks usually works by back-propagating gradients of the loss through the model. During weight optimization these gradients (with respect to the parameters) are used to minimize the loss by adjusting the weights.



However, the gradients can also be calculated with respect to the input itself. Consequently, the **input of a Neural Network can be modified to cause a desired behavior in the fixed model.**



These techniques can be called **Input Optimization**. The lecture deals with two instances, namely **Adversarial Attacks** and **Neural Style Transfer**.

### ADVERSARIAL ATTACKS

**Adversarial Attacks** are **techniques to deliberately cause malfunctions in prediction model**.

More specific, **Adversarial Examples** are **intentionally created instances to deceive the model**.

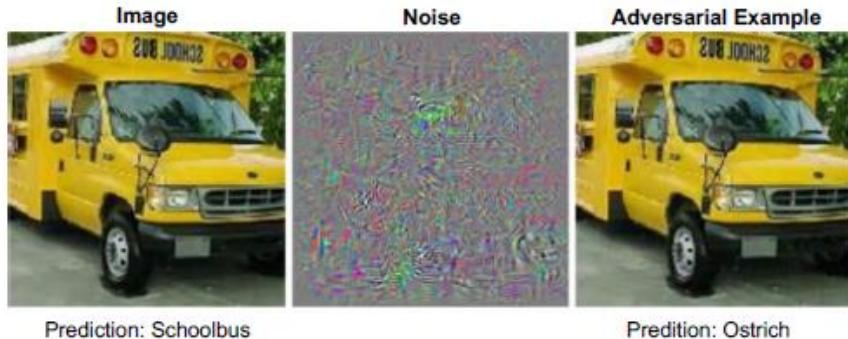
One way to create them is using Input Optimization.

One of the first examples was a technique called **L-BFGS-Attack**. The idea is to take a fixed classifier  $f: \mathbb{R} \rightarrow \{1, \dots, L\}$  and an image  $x$  that was correctly classified by  $f$ . The adversarial example is generated by adding a small portion of noise to the image  $x$ , so that:

$$x + \arg \min_{\Delta x} \{\|\Delta x\|_2 : f(x + \Delta x) = y_t\}$$

Where  $y_t$  denotes the target class. The new image is then classified to any  $y_t$  with a high confidence in the softmax distribution. Furthermore, the noise is quasi-imperceptible for a

human observer. L-BFGS-Attack demonstrated that the classification of images (with CNNs) is remarkably fragile.



Additionally, it has been demonstrated that adversarial perturbations are also able to deceive semantic segmentation models.

Generally, these attacks are **using direct access to the system and the exact input image**. Consequently, the danger of these methods was not broadly recognized. The attacks were underrated they only work models-specific and when the gradients information is available. Furthermore, the attack only works when concrete input pixels are manipulated. An attack in a real-world environment does not have such constructed circumstances.

### *Robust Adversarial Attacks*

However, newer research demonstrates that so called **Robust Adversarial Examples** exist. It has been shown that classifiers in the physical world can be attacked, by constructing examples that not only work for static images. This is done by maximizing the **expectation over transformation**  $\tau$  (EOT):

$$\arg \max_{x'} E_{t \sim \tau} [\log P(y_t | t(x')) - \lambda \|t(x') - t(x)\|_2]$$

with a set of possible transformations  $t$  and for a target class  $y_t$ . Intuitively, the attack is constructed to cause malfunctions for different perspectives and viewing angles of the adversarial example. However, it is notable that a larger distribution of transformations  $t$  require larger permutations which eventually can be recognized by humans.

Another possibility is the usage of **unsuspicious** robust attacks that mimic "graffiti" like structures that cause malfunctions in the classifier.

### *Adversarial Patch Attacks*

Another popular technique are **Adversarial Patch Attacks**. In practice they are easy to apply, because the attack consists of a patch that is added to a scene. These patches are optimized across many images to classify the scene as any target class. The attack is robust because the patches were optimized to be effective under a wide variety of transformations.



It has been demonstrated that patch attacks also work for optical flow networks. Let  $F(I, I')$  be an optical flow network and  $\mathcal{I}$  be the data set  $(I, I')$  of an image and the optical flow. Then  $A(I, p, t, l)$  denotes the insertion of a patch  $p$  (transformed by  $t$ ) into the image  $I$  at location  $l$ . Furthermore,  $\mathcal{T}$  is the distribution of affine 2D transformations of the patch and  $\mathcal{L}$  denotes the uniform distribution where the patch is placed in the image. The goal is to find a patch  $\hat{p}$  so that:

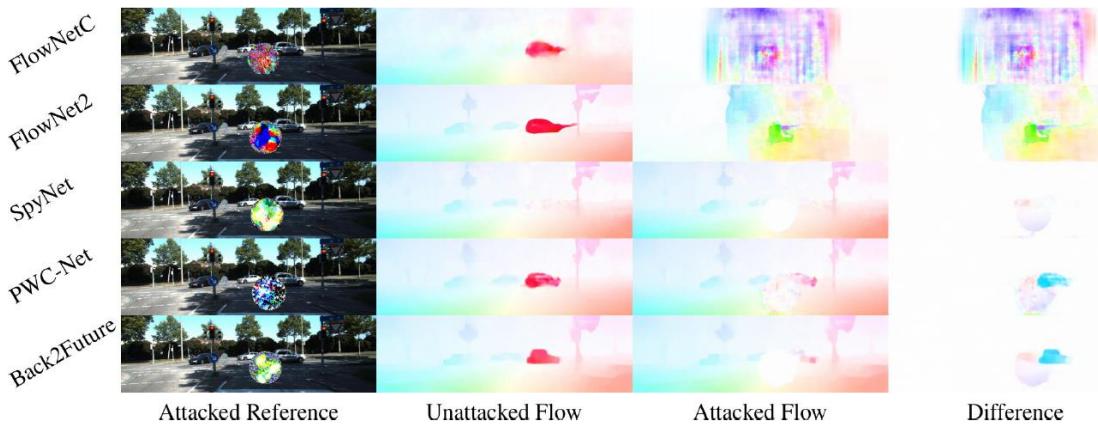
$$\hat{p} = \arg \max_p E_{(I, I') \sim \mathcal{I}, t \sim \mathcal{T}, l \sim \mathcal{L}} \left[ \frac{(u, v) \cdot (\tilde{u}, \tilde{v})}{\|(u, v)\|_2 \cdot \|(\tilde{u}, \tilde{v})\|_2} \right], \quad \text{with}$$

$$(u, v) = F(I, I')$$

$$(\tilde{u}, \tilde{v}) = F(A(I, p, t, l), A(I', p, t, l))$$

Intuitively, the patch  $\hat{p}$  is optimized to reverse the direction of the predicted optical flow.

When the **patches are optimized for a specific network**, it is called a **White-Box Attack**.



Generally, larger patches cause a higher disturbance of the predicted optical flow. However, even small patches disturb the prediction far beyond the attacked area. Furthermore, a patch can be trained on a discrete set of networks (e.g. FlowNet2 and PWCNet) and is still **able to disturb the prediction of unseen networks**. This is called a **Black-Box Attack**.

### *Defenses against Adversarial Attacks*

Adversarial Attacks created a whole separate field that deals to defend these attacks. A review from 2019 summarized these defenses into three distinct categories.

- **Gradient Masking/Obfuscation:** Hiding or masking the gradient of the model because most attack algorithms utilize the gradient information to exploit vulnerabilities of the classifier.
- **Robust Optimization:** A manner of learning, so that the model is more robust against adversarial attacks. By relearning and providing adversarial examples, the network thereby becomes more immune against constructed attacks.
- **Adversarial Example Detection:** Methods which first distinguish whether the input is benign or adversarial. Consequently, adversarial instances are rejected in the first place.

## NEURAL STYLE TRANSFER

Input optimization can be used to find vulnerabilities in classifiers. However, the same technique is used to generate new data itself. **Neural Style Transfer** utilized input optimization to synthesize new images. The method receives a content image and a style image. The goal is to **optimize a random image so that it reproduces the content of content image and transfers the characteristics of a style image**. The loss of the network consists of a separate content and style loss:

$$\mathcal{L}_{content} = \mathcal{L}_{content} + \mathcal{L}_{style}$$

Neural Style Transfer uses a VGG network as feature extractor that is pre-trained on ImageNet.

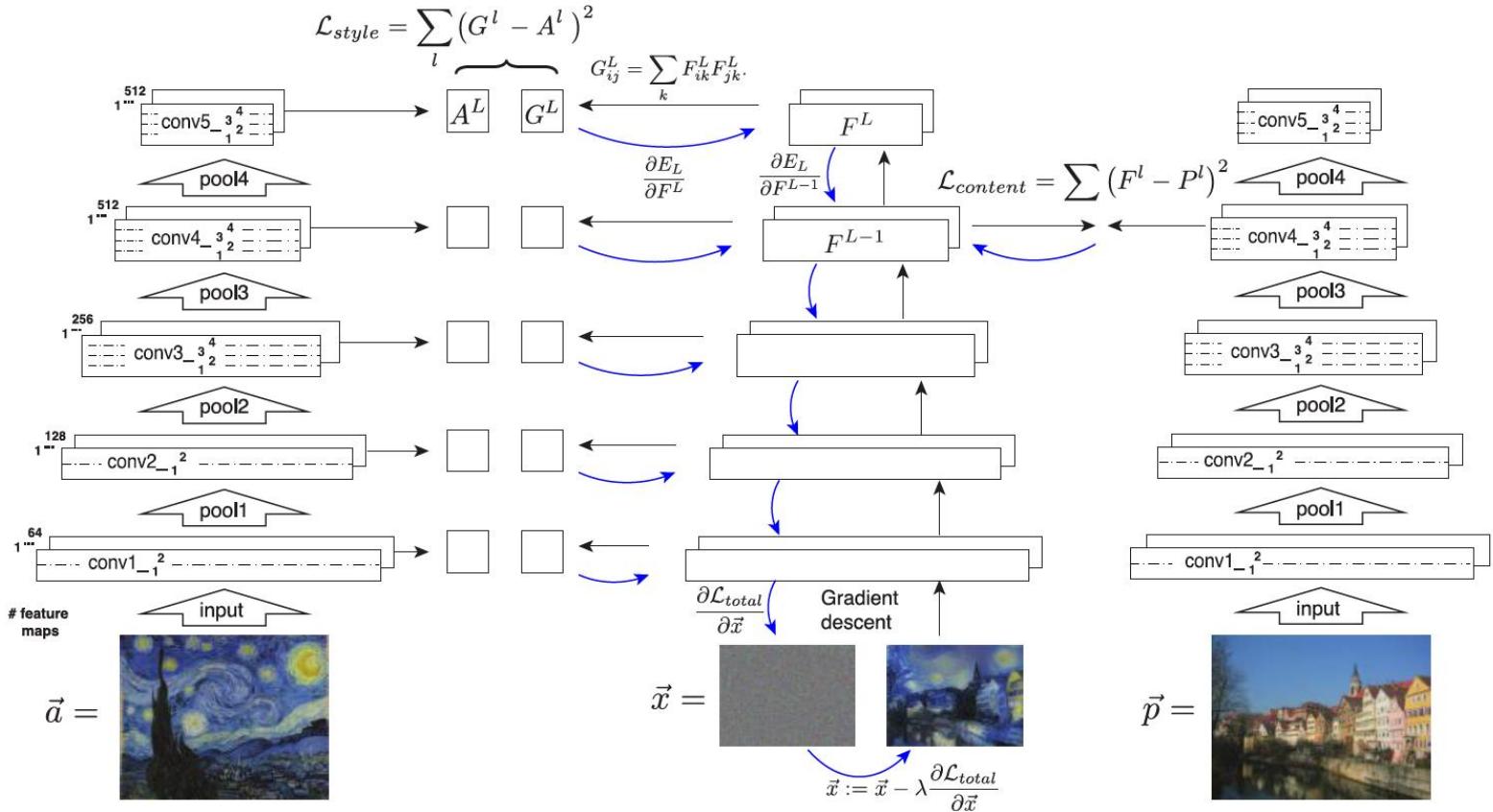


### *Content Reconstruction*

The content image is given to the fixed and pre-trained VGG Net. In the forward pass, the network will generate feature maps  $P_l$  at any layer  $l$  for the given content image. Separately, a random image  $\bar{x}$  (with the same size) is given to the pre-trained VGG net, to generate feature maps  $F_l$ . The content loss

$$\mathcal{L}_{content}(l) = \|F_l - P_l\|_2^2$$

quantifies the difference of the extracted features in a given layer  $l$ . When the content loss is back-propagated, the random image is optimized so that the extracted feature maps are similar in the VGG net. Thereby, the content is reconstructed in the random image.



### Style Reconstruction

The style loss is more difficult but follows a similar principle. The goal of style reconstruction is to **minimize feature correlations of the generated image and the style image**. The reconstruction utilizes the Gram matrices of  $G_l$  and  $A_l$ , for the generated image and style image, respectively. The Gram matrices represent the pairwise correlation of features channels at the given layer. The complete style loss is given by:

$$\mathcal{L}_{content}(l) = \|G_l - A_l\|_2^2, \quad G_{i,j}^l = \sum_{p \in \Omega} F_i^l(p) F_j^l(p)$$

The style loss is combined over several layers of the network. By using the Gram matrices, the content of the style image becomes secondary and almost unrecognizable. However, the artistic characteristics are preserved.

In total, Neural Style transfer uses the gradients of the content loss  $\mathcal{L}_{content}$  and the style loss  $\mathcal{L}_{style}$  to optimize the white noise image  $\vec{x}$ . The optimized image combines the content and style of the given images and yields truly beautiful results.