Discussion 2

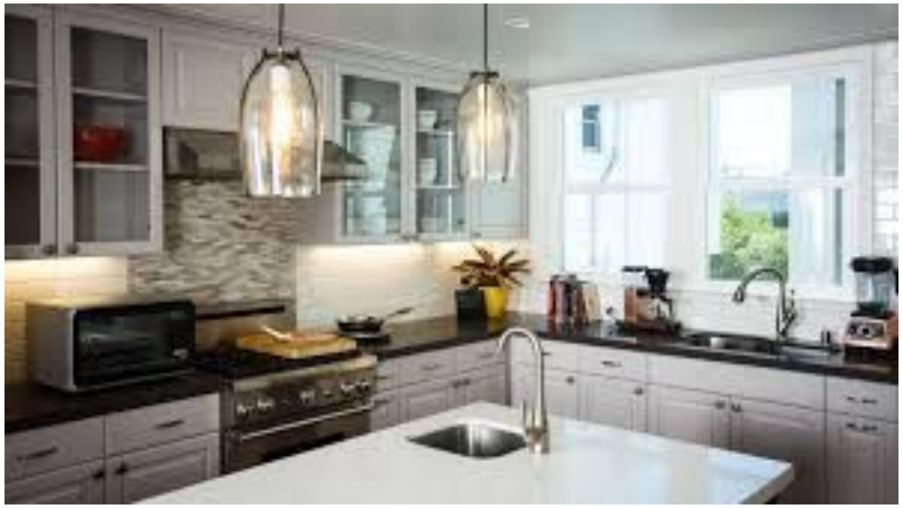
CS188 - Fall 19

Objectives

- Homework 1
  - What is the goal?
  - What are the tasks?
  - Where to start?
- Image Features
  - Why do we need image features?
  - What makes a good image feature?
  - SIFT - Fundamental concept
- Bag-Of-Words Model
  - Why does it work?
- Clustering
  - What is a clustering algorithm?
  - K-Means and Hierarchical Agglomerative Clustering

A visual recognition system

- The idea behind homework 1 is simple: given an image, you will write an algorithm (in Python) that assigns it a ‘label’
- So, given this image:



Your algorithm will output: ???

A visual recognition system

- The idea behind homework 1 is simple: given an image, you will write an algorithm that assigns it a ‘label’
- So, given this image:

Your algorithm will output: kitchen

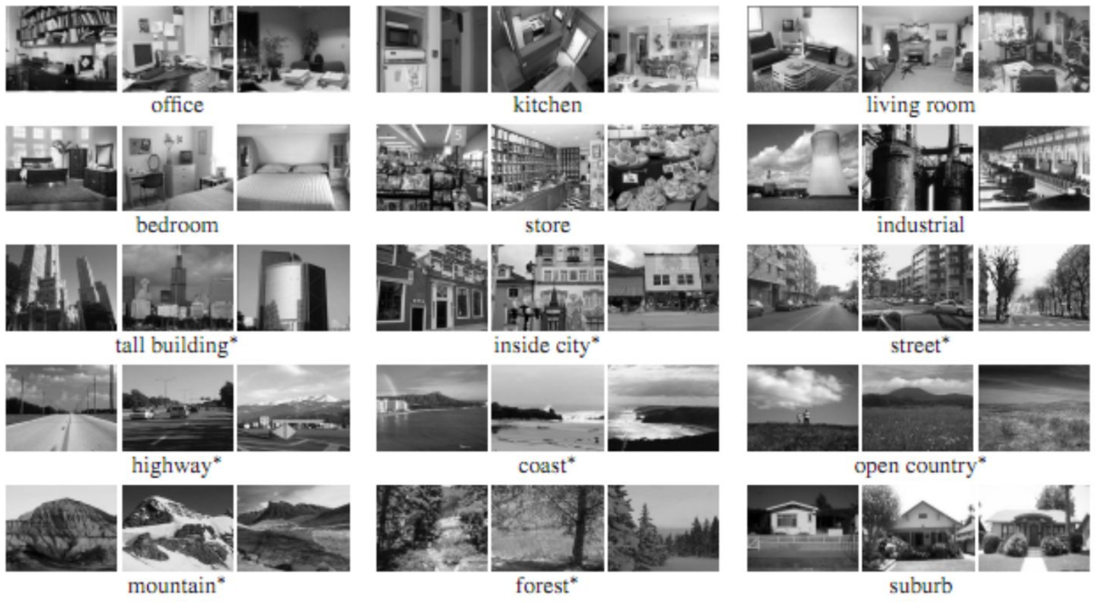
A simple idea, but ...

- To actually get this to work, you will need to display understanding of:
  - Linear algebra
    - Gradients, Laplacians, Hessians, Gaussians, Kernel functions, ...
  - Machine learning
    - Clustering, Support Vector Machines, Nearest Neighbors, ...
  - Computer Vision
    - Feature descriptors, Bag-Of-Words Model, ...

- The goal of this discussion (and the next) will be to go over these topics

Logistics

- Released tonight on CCLE, you will get:
  - A spec, detailing all the steps you need to follow
  - Starter code, to help organize the tasks
  - A dataset, containing 3000 images
- The dataset looks like this (15 categories):



Logistics (cont)

- You are encouraged to work on it in pairs
- You have 2 weeks to complete it
  - The deadline is a **hard** deadline
  - The submission will be on CCLE, **a single tarball per group**
- Your submission will contain:
  - A directory called ‘code’ which contains your python scripts
  - A text file called README that contains your names, UIDs, email addresses and lists the sources you used.
  - You may organize python scripts as you wish **but**
    - One of the scripts must be called homework1.py
    - This script must import all functions from all the other scripts
    - The spec defines the function signatures we want you to implement, feel free to add more
- During grading, we will import homework 1, and call the functions we asked for

A note

- The concepts that you are exposed to are hard to grasp and numerous, please start early!
- That being said:
  - You will not be asked to code any algorithm from scratch
  - In fact the entire project should fit in 200 lines of code, including boilerplate code
- Your time will mostly be spent understanding the ideas behind the algorithms, not programming
  - This involves searching through APIs
  - This involves doing some research on the topics covered
- These ideas are extremely important, and very useful in the real world
  - This homework is not aimed at keeping you busy

The tasks

- Write a nearest neighbor classifier using Tiny Image features
- Implement a Bag-Of-Words representation for images
- Write a nearest neighbor classifier using a BOW representation, that uses:
    - SIFT features
    - SURF features
    - ORB features
- Write a linear SVM to classify data using the BOW model
- Write a nonlinear SVM to classify data using the BOW model

A look at the spec

Getting started

- In a first pass, try to solve all sections of the project, with the following exceptions:
  - Ignore the effect of the number of neighbors during classification
  - Using only SIFT features and K-means for the BOW vocabulary
  - Using only linear SVMs with SIFT features for the last part
- Why I recommend you do it this way:
  - I'll talk about kernel-SVM, ORB, SURF next week (Bad reason)
  - You'll be building a full pipeline this way (good reason)
    - If you solve this the coming week, you'll just be solving the **same** problem using **different** methods next week
    - You'll already have a good idea of all the steps
- Why are you solving the same problem differently?
  - There is no 'optimal' in machine learning, everything depends on your task and dataset: versatility is important

Why do we need feature descriptors?

- Computers see the world through images, which are arrays of integer values
  - Note that the world is 3D, images are only 2D
  - How can a computer differentiate a scene from a picture of a scene ..?
- If I ask you what room we're in right now:
  - People in the front right would say: a classroom
  - People in the back left would say: a classroom
  - If I dimmed the lights, you would still recognize a classroom
  - If you tilt your head to the side, you would still recognize a classroom
- Yet all of those differences incur a huge change to our array of pixels!
  - How is the computer supposed to recognize 'classroom' ?

An array of pixels

- This isn't a good 'representation' of the scene for the task of recognition
    - You wouldn't describe the classroom to someone you're talking with on the phone by giving them a description of the wall colors, inch by inch ...
- What would we like instead? Invariance w.r.t:
    - Illumination:
        - So that if I take two pictures with different lighting, the scene is still recognized as a classroom
    - Scale/Orientation:
        - If I take a picture from the back or the front while crooked, the scene is still a classroom
    - Viewpoint change:
        - If I move in the scene and take another picture, it is still recognized as a classroom

Image features

- Instead of describing the image pixel by pixel, we would like to move closer to semantics, where we would describe an image object by object
    - Object detection/recognition is really hard …
- Features are an intermediate step: we can describe an image as a set of features, that are hopefully invariant to rotation, illumination changes, scale, …
    - A feature is a small patch in the image, made up of 2 parts:
        - A keypoint: 2D position of the patch, its orientation, scale, …
        - A descriptor: Contains the visual description of the patch
    - If I define my features ‘well’, I could :
        - take a picture of this classroom, dim the lights, move around
        - take another picture: both would have similar features
        - I could therefore ‘match’ both images as representing the same scene

How to find them?

- This is a question that has been explored by researchers over the past 40 years
- They came up with a lot of methods (Harris corner detector, FAST corner detector, BRIEF, MOPs, SIFT, SURF, ORB, ...)
- Recently, Deep Learning efforts focused on the problem
- By far the most influential is the SIFT feature, which is a fundamental idea in computer vision
- Let’s explore how SIFT works (at a high level)

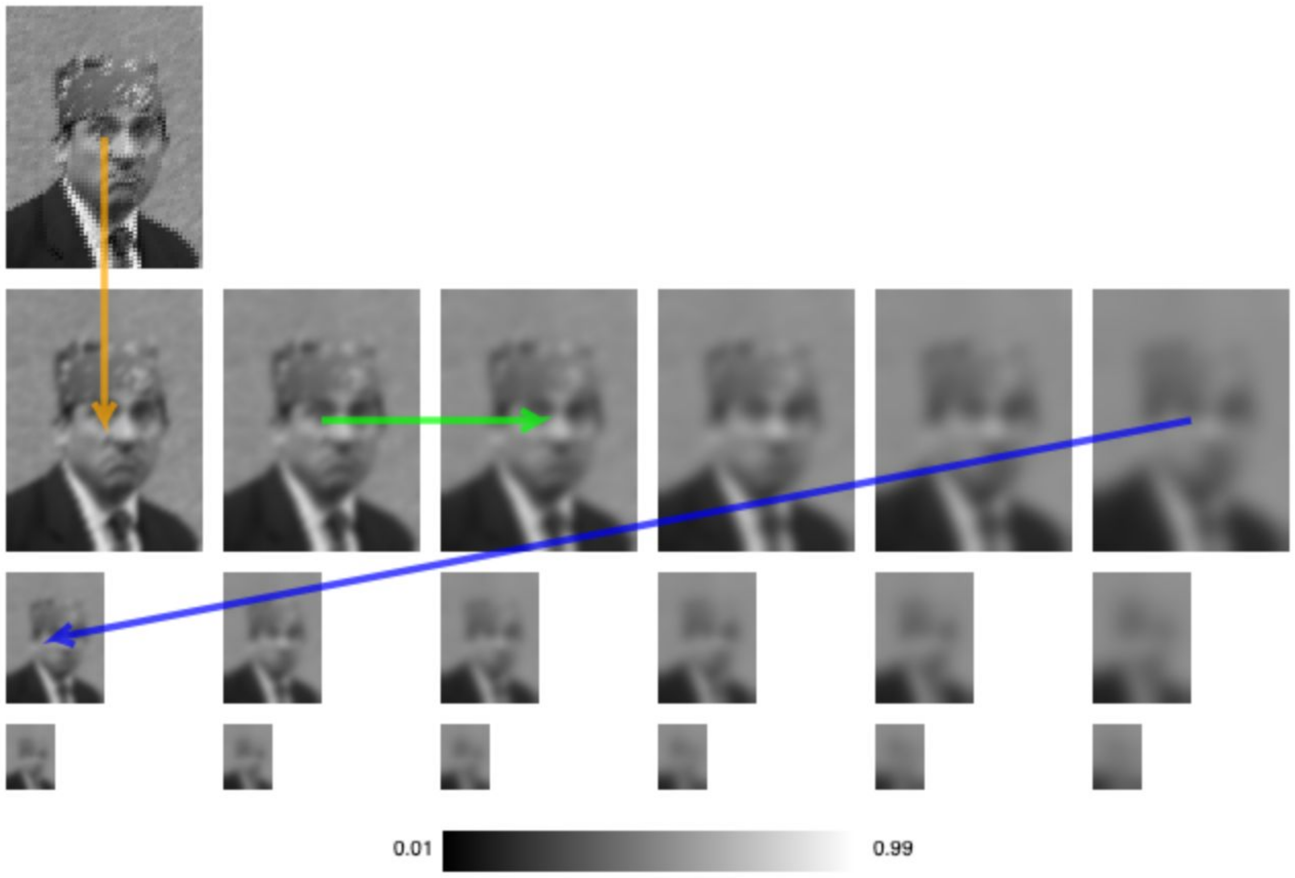
SIFT through examples

- Remember, the goal is to find image patches that are invariant to :
    - Illumination change
    - Viewpoint change
        - Location/Rotation
    - Scale change

Step 1: Feature Point Detection

- The first step is to find a lot of patches in the image of **potential** interest
  - We will remove a lot of them in later steps
- For now, we worry about scale invariance:
  - We would like patches that can be found from up close and far away
- But we only have one image, how do we simulate different distances?
  - Upsampling: Zooming-in
  - Blurring + downsampling: zooming-out
    - Note that we need to blur first, to remove high-frequency information
  - Using these operations, we can create a scale-space
    - A 3D space!

Step 1: The scale space

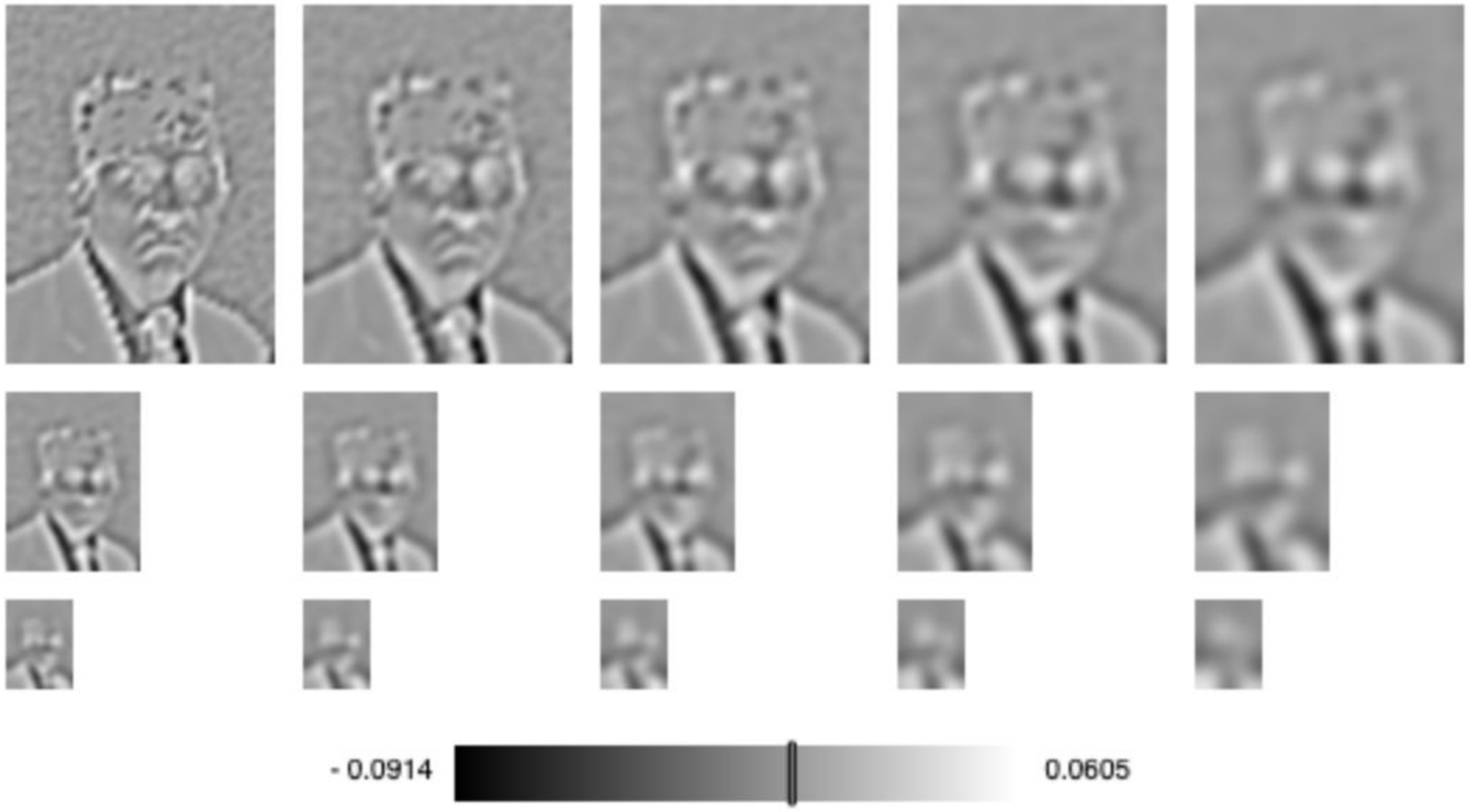


If I could find a patch that stayed relatively constant in all these images, it could be considered ‘scale invariant’ (Goal 1 of 3)

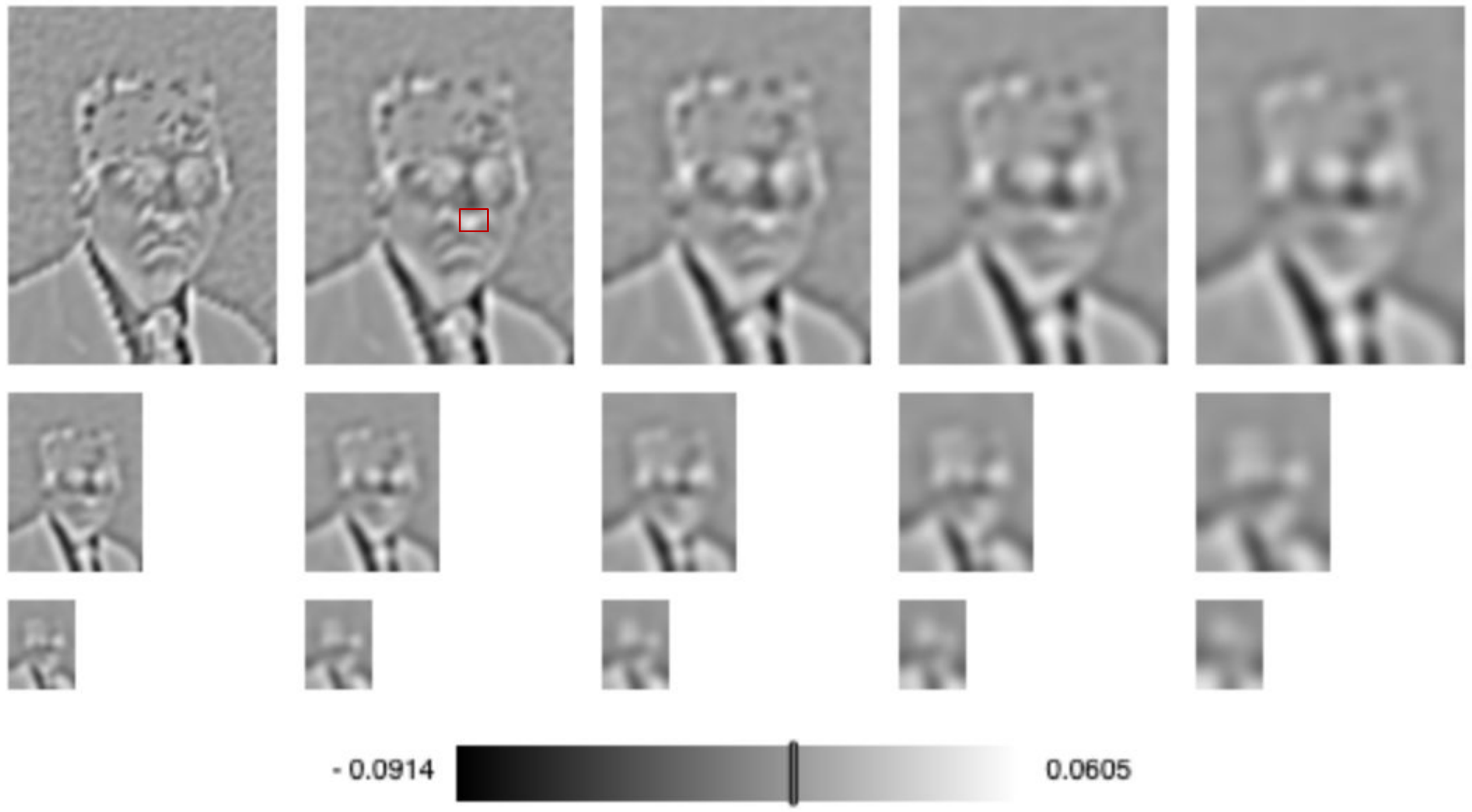
Step 1: Difference of Gaussians (DoG)

- We mentioned that comparing pixel values to each other directly was not a good idea
  - We're going through this whole process to avoid doing exactly that
- Instead, we'd like to detect edges that remain present at different scales
- The DoG (discrete) approximates the Laplacian (continuous)
  - The Laplacian is a 'blob' detector
  - If a 'blob' can be detected at any scale, it is scale-invariant
  - You can think of it as an edge detection for edges of a particular scale
  - DoG is a bandpass filter, finding an edge for every 'band' indicates that the current patch could be spotted from any scale

Step 1: DoG



Step 1: DoG



Step 1: DoG (Zoom)

The difference for a pixel is computed as such:

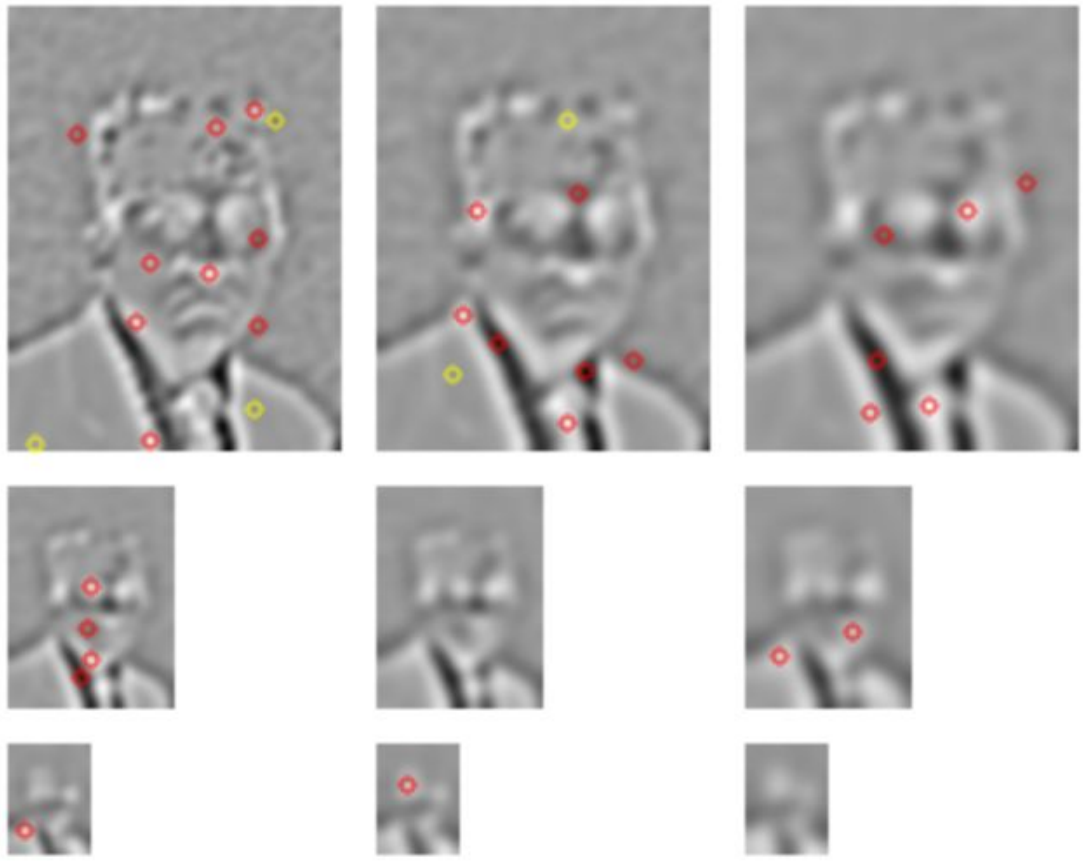
0.311
0.039
0.349

Step 1: Scale Space Extrema

- Extrema in the scale space are good candidates
  - They represent edges that are the most robust to scale change

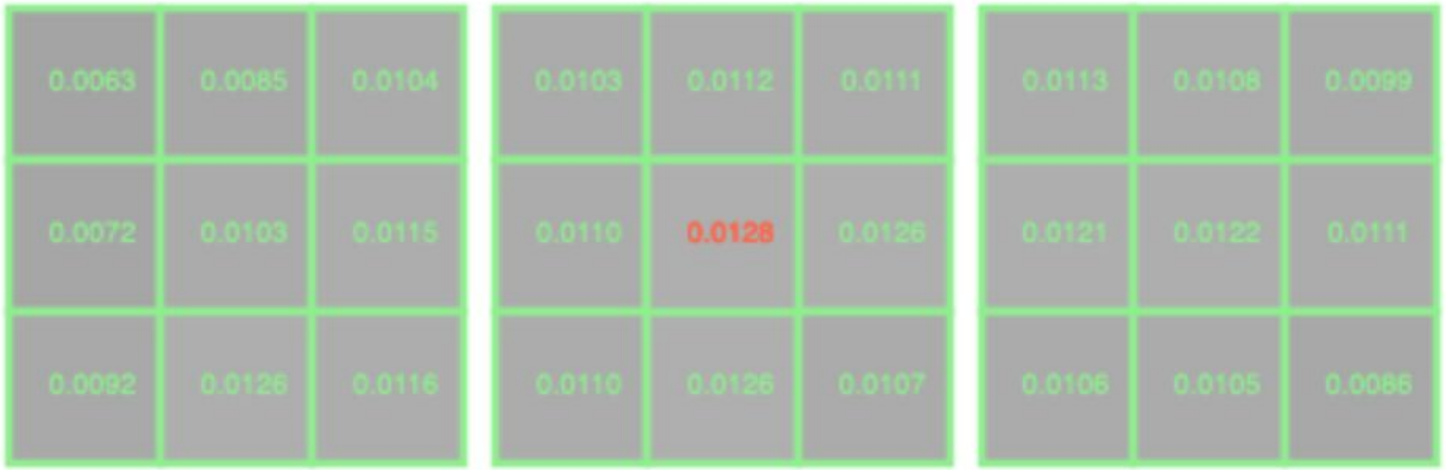
- An extrema, in this case, is calculated using 26 neighbors:
  - 8 in the current image
  - 9 in the image ‘below’ (smaller Gaussian std)
  - 9 in the image ‘above’ (larger Gaussian std)

Step 1: Some potential features



Step 1: Calculation example

The patch on the bandana:



0.0063	0.0085	0.0104	0.0103	0.0112	0.0111	0.0113	0.0108	0.0099
0.0072	0.0103	0.0115	0.0110	0.0128	0.0126	0.0121	0.0122	0.0111
0.0092	0.0126	0.0116	0.0110	0.0126	0.0107	0.0106	0.0105	0.0086

Step 1: Conclusion

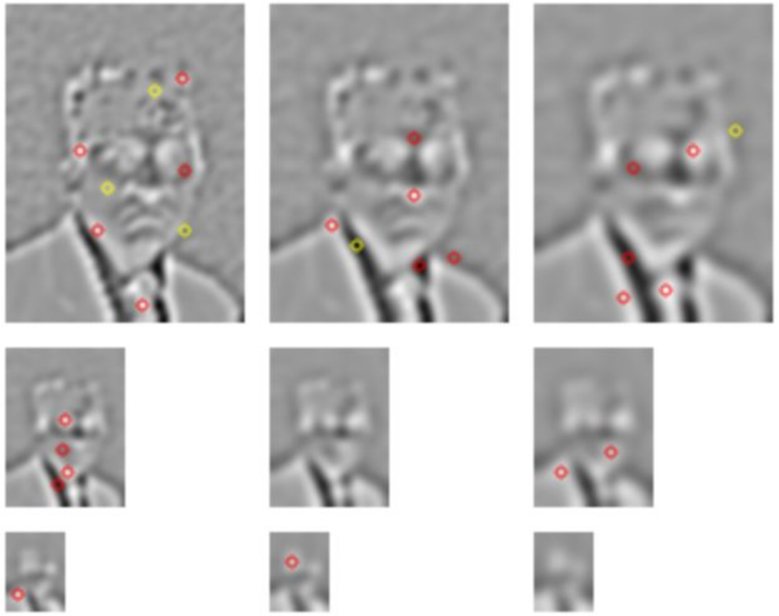
- We found patches that contain edges that are robust to change of scale
- We discarded extrema with small absolute values, considering that they are due to noise
- The patches we are considering are now scale invariant, which was one of 3 goals.
- Let’s move on to step 2

Step 2: Coordinate Refinement

- Discretization implies loss of precision
- The algorithm uses a Taylor expansion to refine the coordinates
    - Don’t worry about this for now
    - The idea is that due to rounding errors, the extrema in scale-space we found in step one may be off by a pixel in some direction, and this step accounts for that

- The result is a new set of keypoints, close to those in step 1 (moved by a pixel, if at all)

Step 2: Coordinate Refinement



For the nose:

	discrete	interpolated
x	29.0	29.048
y	38.5	38.382
scale	1	1.559

Step 2: Pruning keypoints

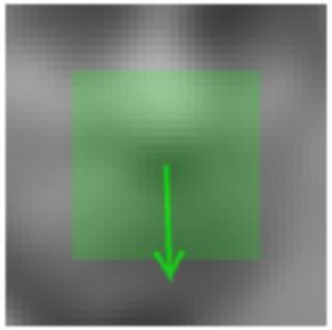
- Key points that lie on an edge are invariant to translation along that edge
  - This is bad for us, we would like to be able to identify the position of the keypoint with certainty!
- These keypoints are therefore removed from the set
- This involves comparing the principal curvatures of the scale space function
  - If there are strong variations in several directions, there are several edges ...
  - This is done through the calculation of the trace and determinant of the Hessian

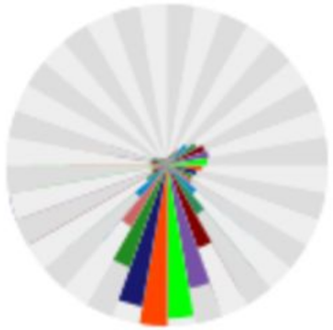
Step 2: Conclusion

- We shifted keypoints by refining their coordinates
  - They are now more precisely identified in **scale-space**
- We pruned keypoints that lied on edges
  - They were invariant to translation along the edge direction
  - and were therefore unreliable

Step 3: Orientation Assignment

- Another goal we had was to be invariant to viewpoint change
  - The first step to that is to assign an orientation to the point
  - For that, we compute gradients in the keypoint's neighborhood, and check their overall direction
  - If all gradients have a similar direction, this becomes the orientation of the keypoint
  - Otherwise, the keypoint is discarded
- For Michael's nose:





Step 3: Why this matters

- When comparing 2 patches, I can now **align** them given their orientation
- I can now counteract the effect of rotations on the image!
  - This was a second goal to define a good feature descriptor

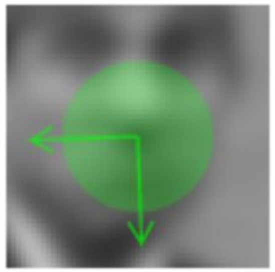
- In conclusion:
  - We now have precisely mapped keypoints, that are invariant to scale change and orientation change
  - Image features are made up of keypoints and their descriptors
    - We now have the keypoints
    - And can move on to Step 4

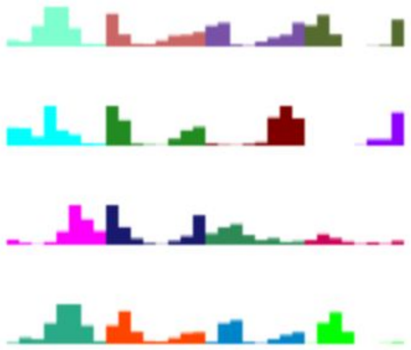
Step 4: Descriptor Generation

- We now know which image patches are interesting
  - We know where they are located
  - And how to orient them for comparison
- How should we describe them?
  - Not with pixels …
- This is similar to step 3: we compute a histogram of gradients over a neighborhood
  - Note that gradients are invariant to shifts in pixel value!
  - They are therefore invariant to illumination change
  - We hit our 3rd (and final goal)

Step 4: Example Descriptor

For Michael’s nose, we actually have 16 histograms!
One per point near the keypoint descriptor





SIFT: Conclusion

- Key points are extracted at different scales and blur levels and all subsequent computations are performed within the scale space framework. This will make the descriptors invariant to image scaling and small changes in perspective.
- Computation relative to a reference orientation is supposed to make the descriptors robust against rotation.
- The descriptor information is stored relative to the key point position and thus invariant against translations.
- Many potential key points are discarded if they are deemed unstable or hard to locate precisely. The remaining key points should thus be relatively immune to image noise.
- The histograms are normalized at the end which means the descriptors will not store the magnitudes of the gradients, only their relations to each other. This should make the descriptors invariant against global, uniform illumination changes.

From image features to Bag-Of-Words

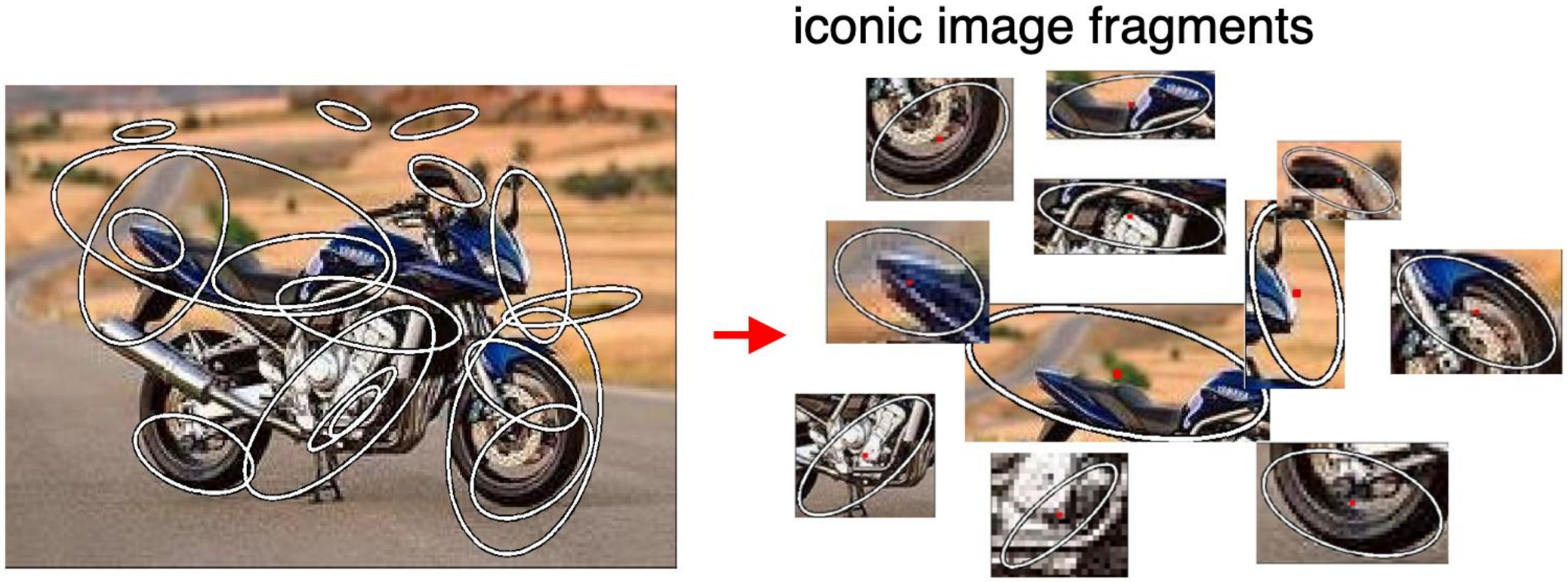
- The hard part is over! BOW is straightforward
- A SIFT descriptor is a way to recognize the same point **in the scene** (3D) from different images (2D) with different scale, illumination, viewpoint
- It describes an image **patch**
- A single SIFT feature is not enough to represent an image - it just contains local information
    - How do we represent the entire image with SIFT?
    - Take a lot of image features! Within a single image, extract hundreds of SIFT features
- Using this, how do we compare images?
    - If most of their features match, the images are probably related

Bag-Of-Words

- Words here are “visual”
    - BOW actually comes from Natural Language Processing, text documents would be represented by their frequency of use of certain words
    - With images, “words” are image features (eg: obtained through SIFT) - ie ‘iconic’ image patches

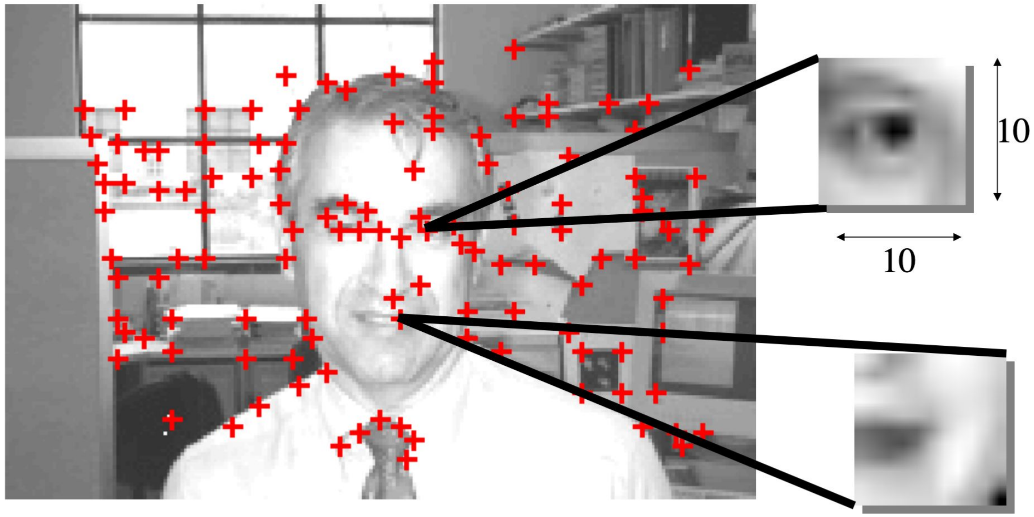
- We do not want them to be ‘ordered’
    - The order in which the features appear/are detected should not matter
    - Therefore all the features are recorded in random order, like you would place them in a bag

In practice



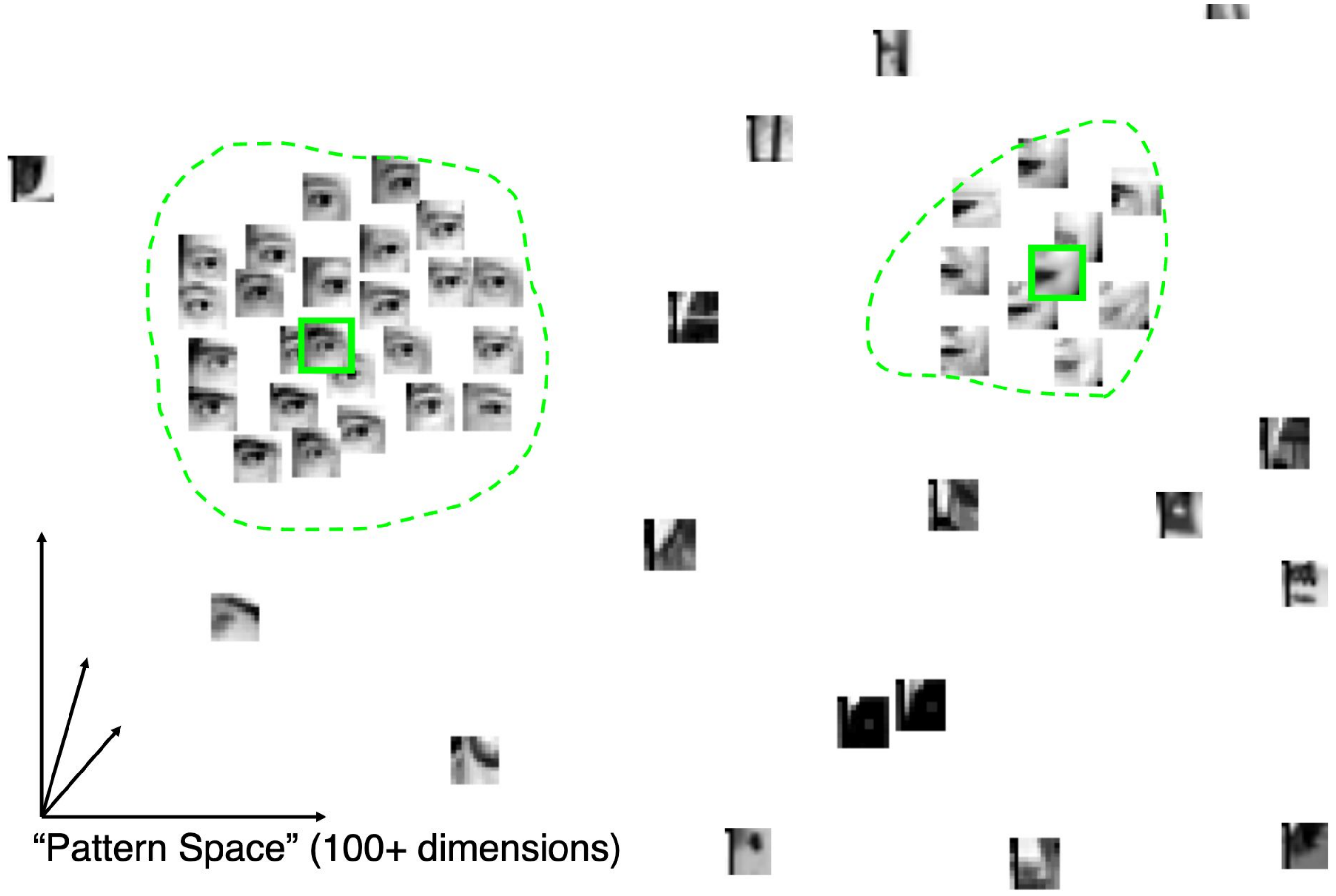
Vocabulary

- You will build a set of features that you can compare every image against
  - This is your **vocabulary**
- To do so, you will extract 100’s of features from different images, and **cluster** them together
  - That way your vocabulary will contain only relevant features that occur frequently
- An example with faces:



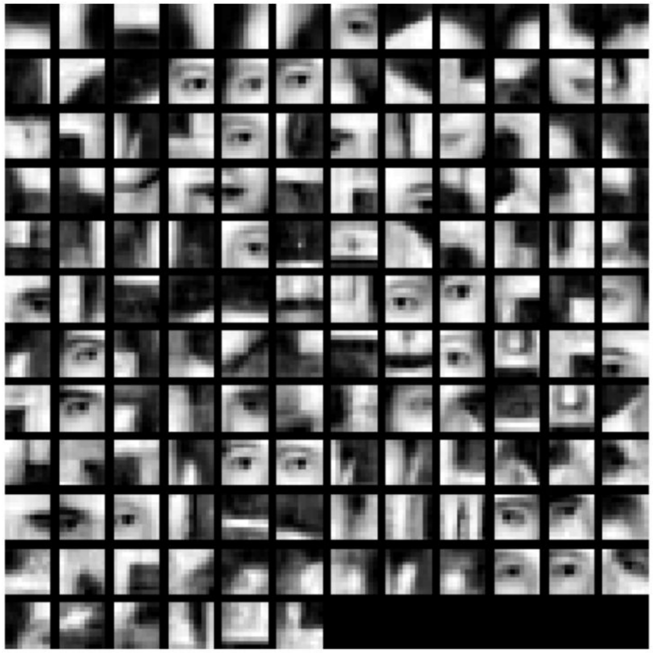
Vocabulary

I can do the same process for several images:



Vocabulary: representation

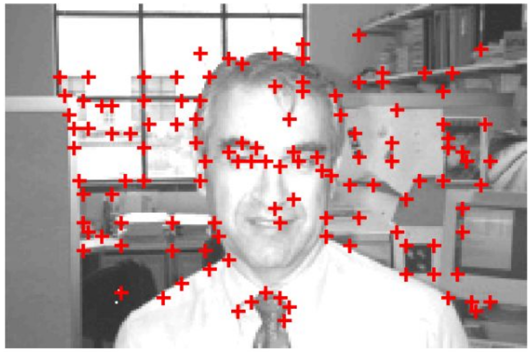
100-1000 images

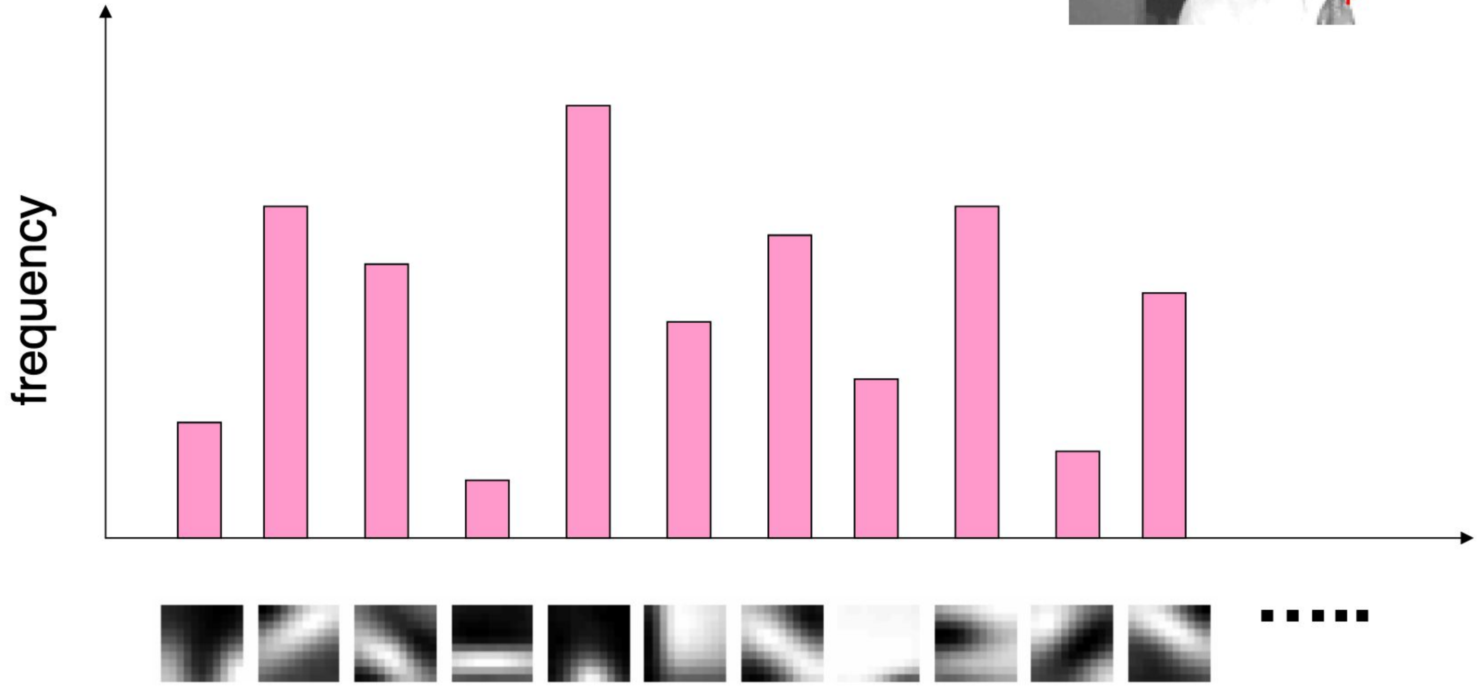


~100 visual words

The BOW representation of an image

- detect interest point features
- find closest visual word to region around detected points
- record number of occurrences, but not position





How is this useful for classification?

- For your homework, you have categories like streets and forests
- Step 1: Represent all images in your training set using the BOW model
  - Your image will now be a vector of features
  - Hopefully, pictures of streets will share a lot of similar features, that are different from the features forests share
- Step 2: Train a classifier
  - What does the histogram of a street picture look like? What does the histogram of a picture of a forest look like? They won't be represented by the same features
- Step 3: Apply the classifier to the test image
  - Extract its BOW representation
  - Does its histogram look like that of a forest, or a street?

Conclusion

- An image as an array of pixels is not very suited for scene recognition, as small changes greatly affect the array
- We would like to represent the image as a collection of features, that are invariant to illumination, scale and viewpoint changes
    - Such features would be considered optimal
- There are several ways to compute such features, SIFT is the most famous
- This allows us to represent scenes in a more ‘semantic’ manner