Large Scale of Vehicle Search and Re-Identification

Seven

Research Scientist
www.julyedu.com

Seven@julyedu.com

April 15, 2018

Preliminary Background

- Basic knowledge of supervised learning (e.g. SVM) and usupervised learning algorithms (e.g. K-means).
- Basic knowledge of neural networks, e.g. feed-forward and backpropagation.
- Experienced with training and finetuning ConvNets.
- Basic level of a programming language, e.g. Python.

In this lecture, you will learn

Preliminary Background

- Basic knowledge of supervised learning (e.g. SVM) and usupervised learning algorithms (e.g. K-means).
- Basic knowledge of neural networks, e.g. feed-forward and backpropagation.
- Experienced with training and finetuning ConvNets.
- Basic level of a programming language, e.g. Python.

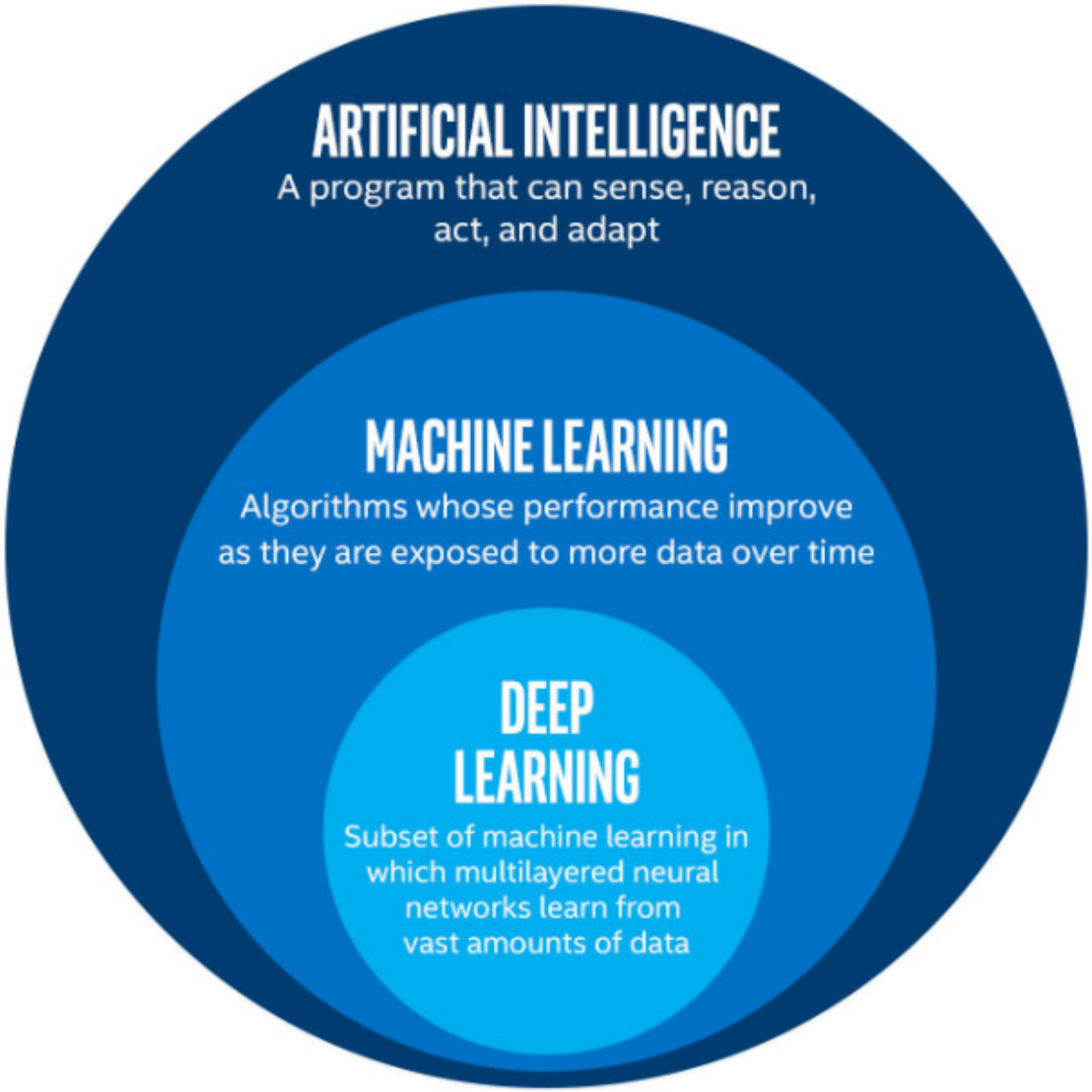
In this lecture, you will learn

- Bag-of-visual-words model for visual representation.
- Multi-task learning framework of ConvNets.
- General image retrieval framework based on CovNets features.
- STOA deep metric learning, i.e. triplet loss or deep ranking.
- Triplet neural networks training tricks including "hard" triplet sampling strategies.
- How to apply triplet loss based ConvNets to obtain STOA results on VehicleID dataset.

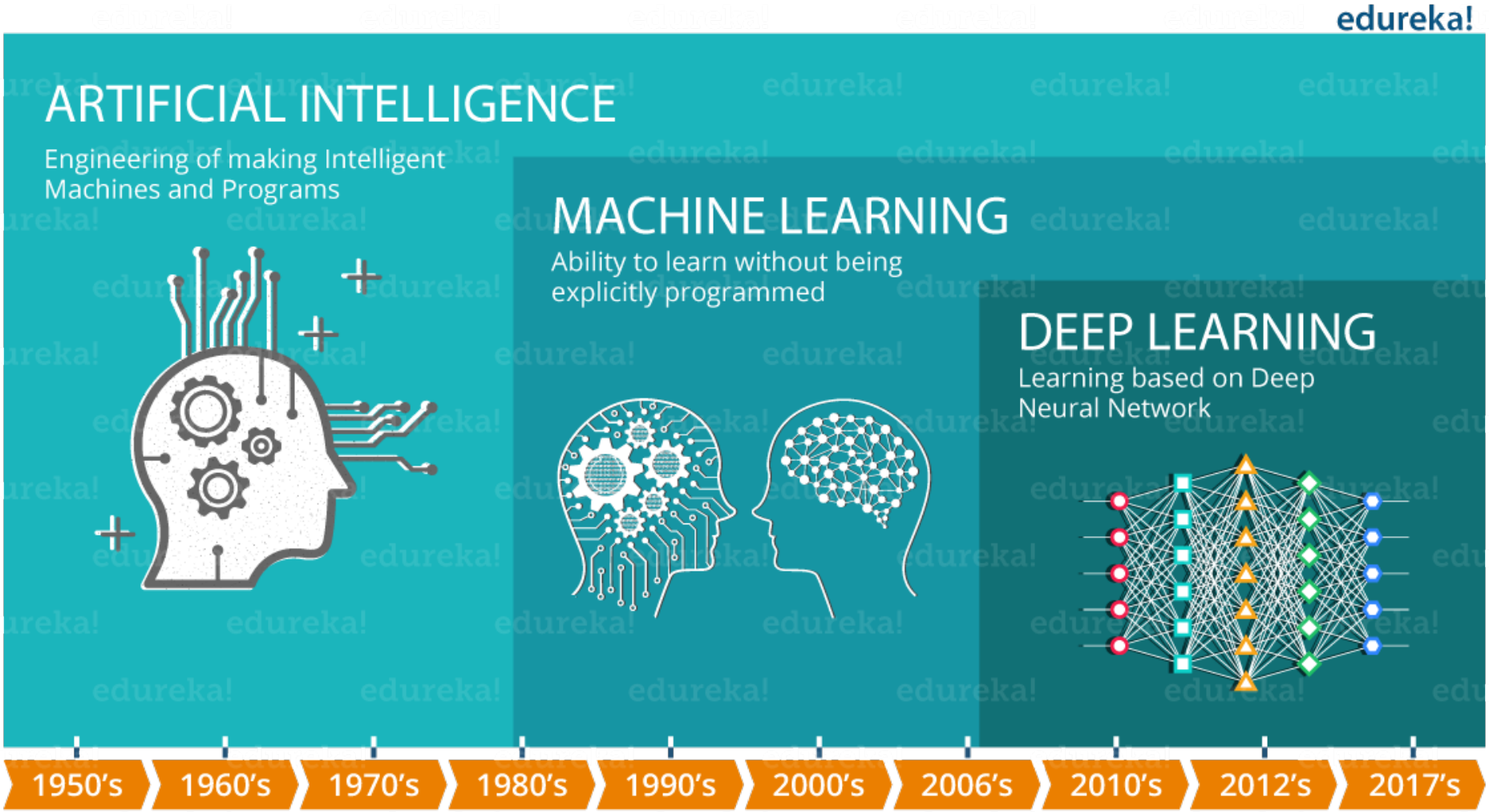
Overview

1. The Key Problem in AI: Feature Extraction
   - Artifical Intelligence, Machine Learning and Deep Learning
   - Machine Learning VS. Deep Learning
2. Visual Representation Before and After 2010
   - Bag-of-Visual-Words: The Landmark of Computer Vision in 2000-2010
     - Bag-of-Words Model for NLP / Text Mining
     - Bag-of-Visual-Words for Computer Vision
   - Deep Learning for Image Retrieval Since 2010
     - Autoencoders for Image Retrieval
     - Extracting Deep Features by ConvNets
   - Tips for Training ConvNets
3. Large Scale of Vehicles Search and Re-Identification
   - Triplet Loss
   - Apply Triplet Loss Network to Solve Vehicle Re-Identification

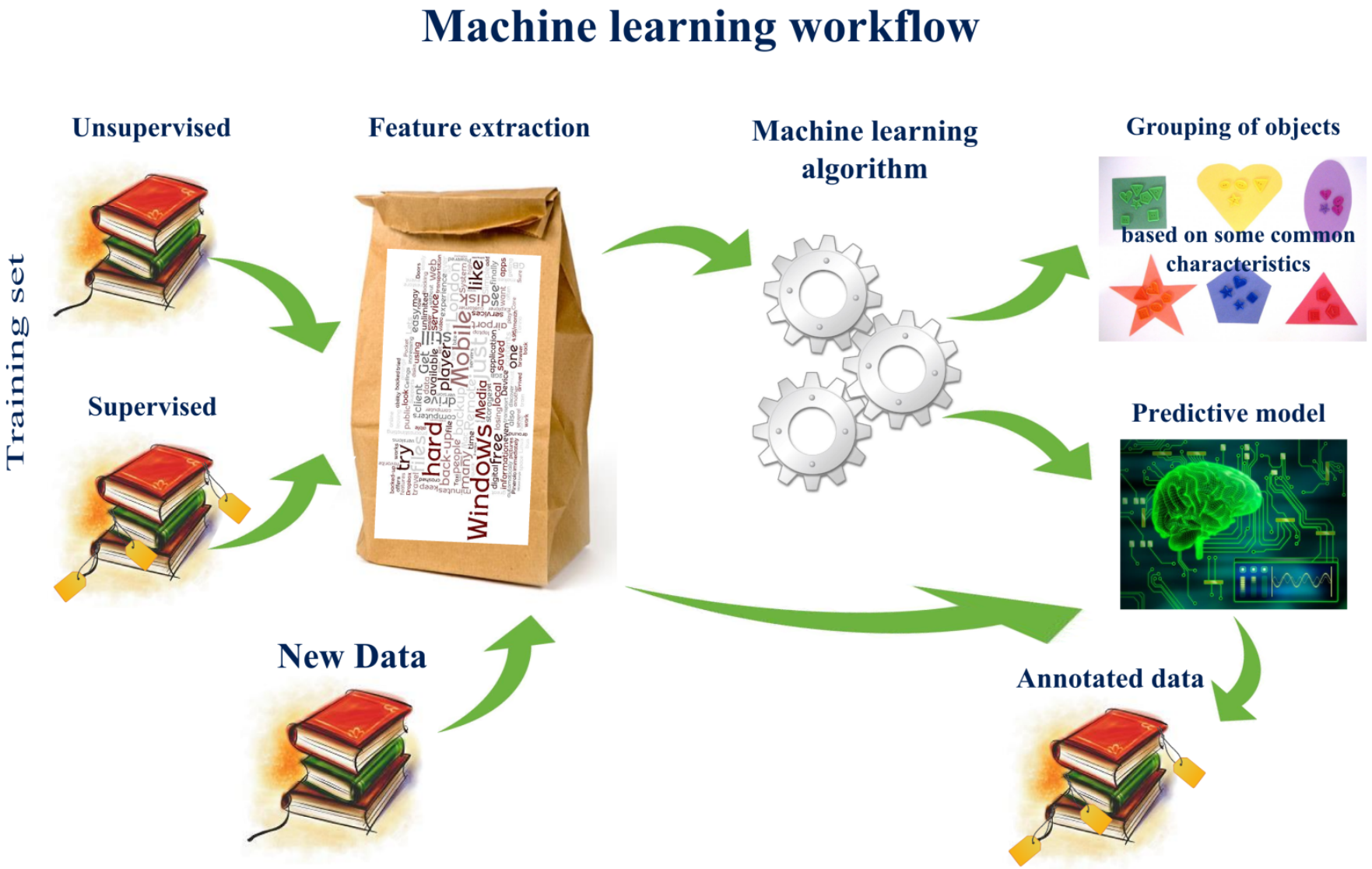
Artifical Intelligence, Machine Learning and Deep Learning



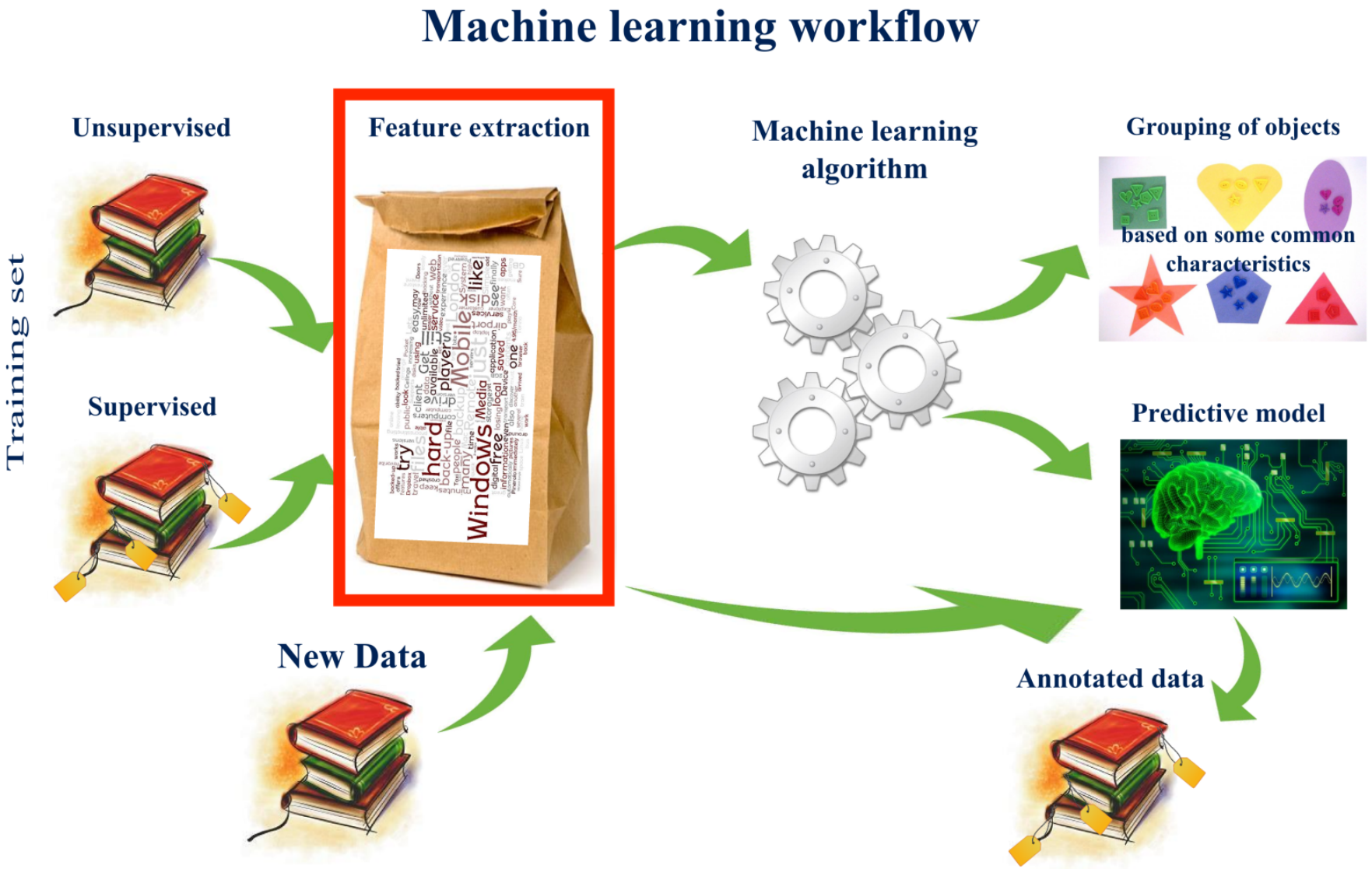
AI Technologies Timeline



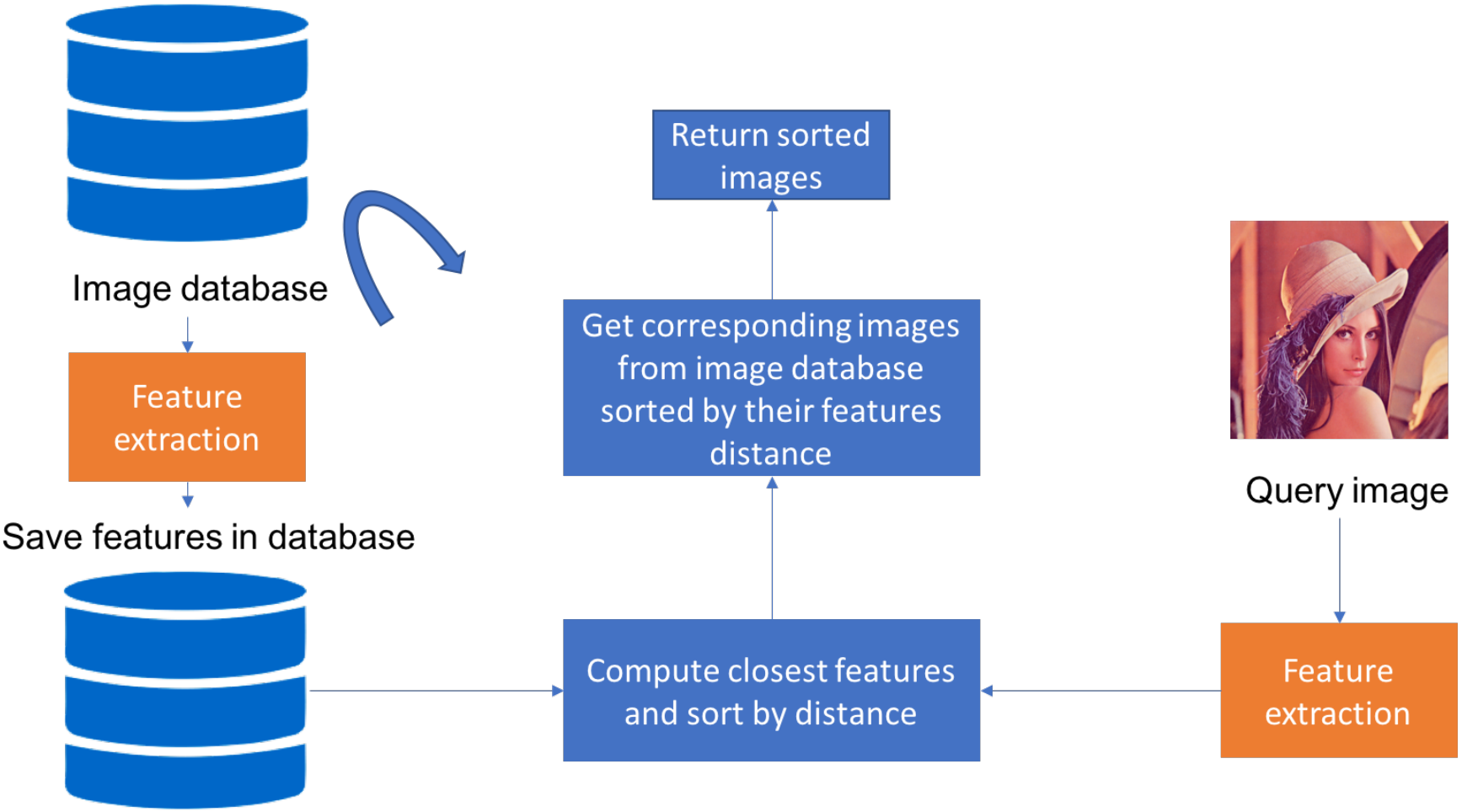
Machine Learning Workflow



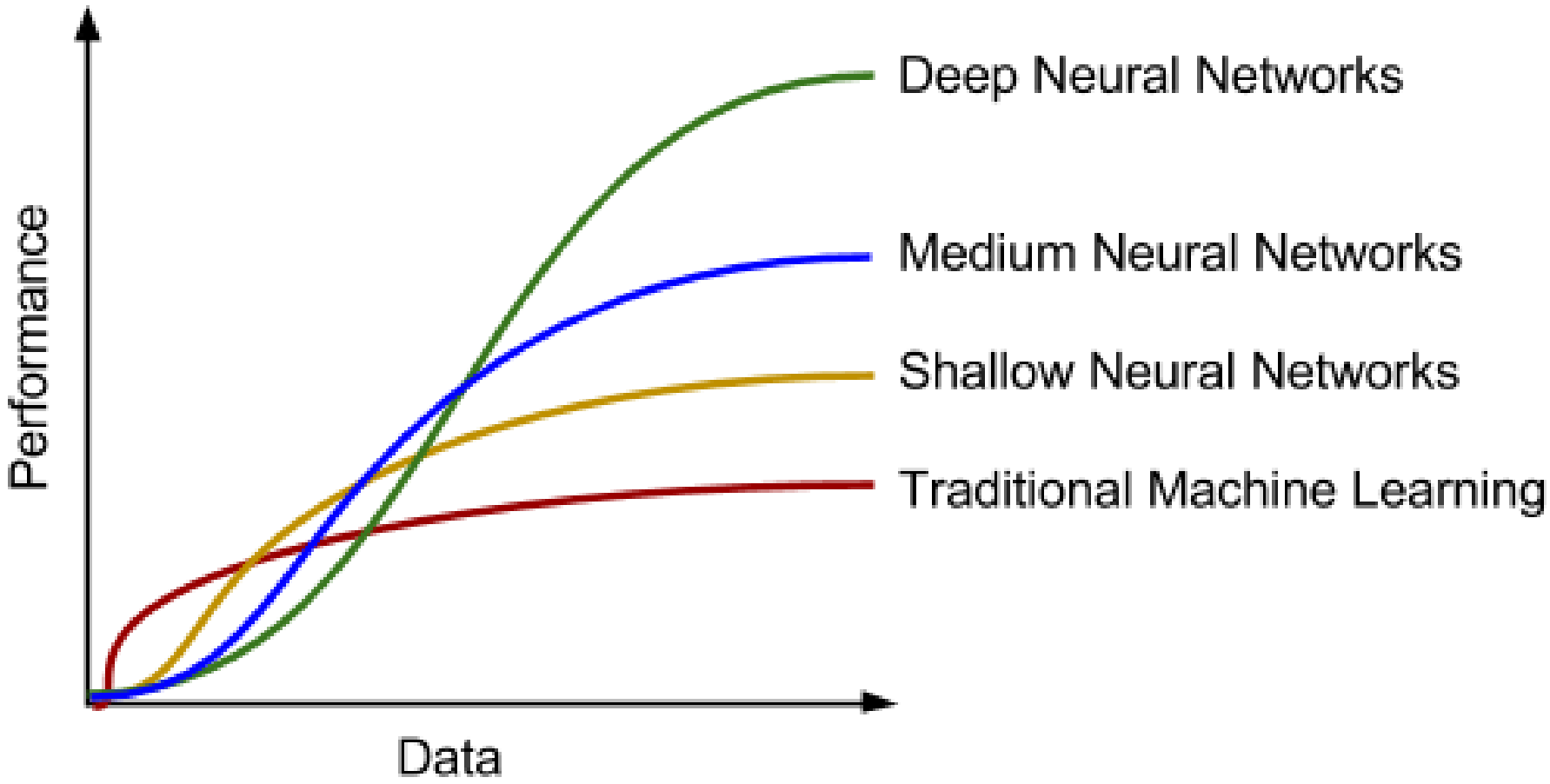
Feature Extraction is The Key



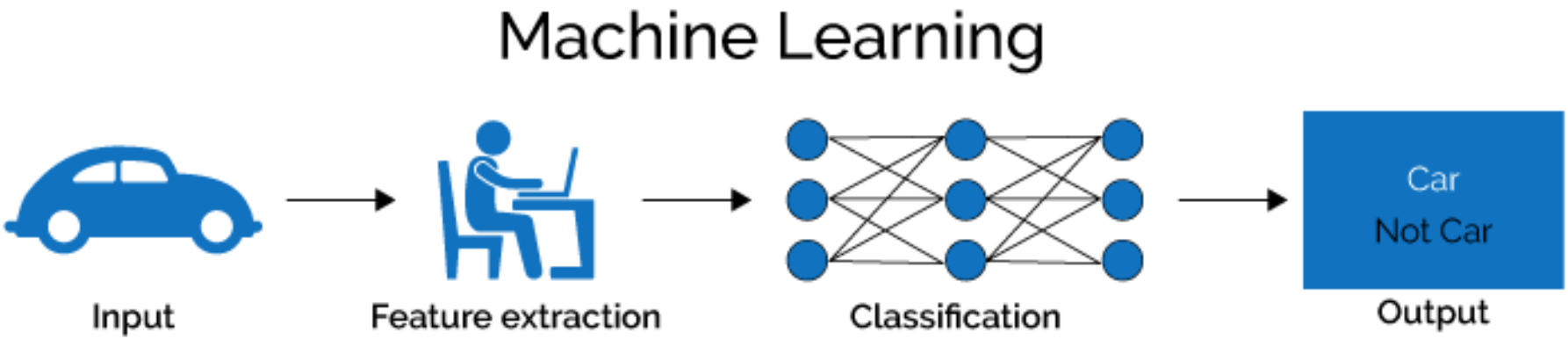
Content-based Image Retrieval (CBIR) Framework

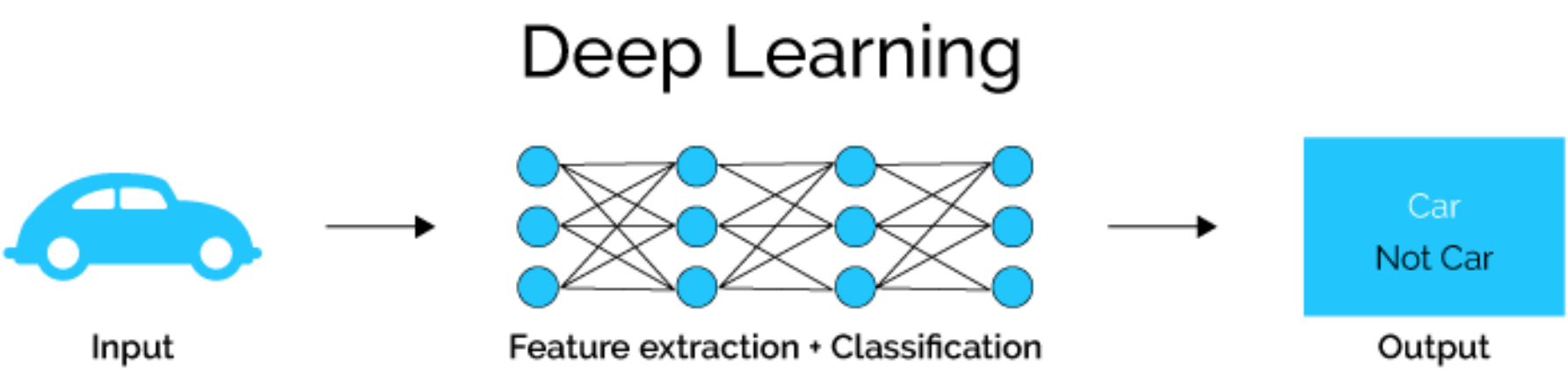


Drawbacks of Traditional Machine Learning

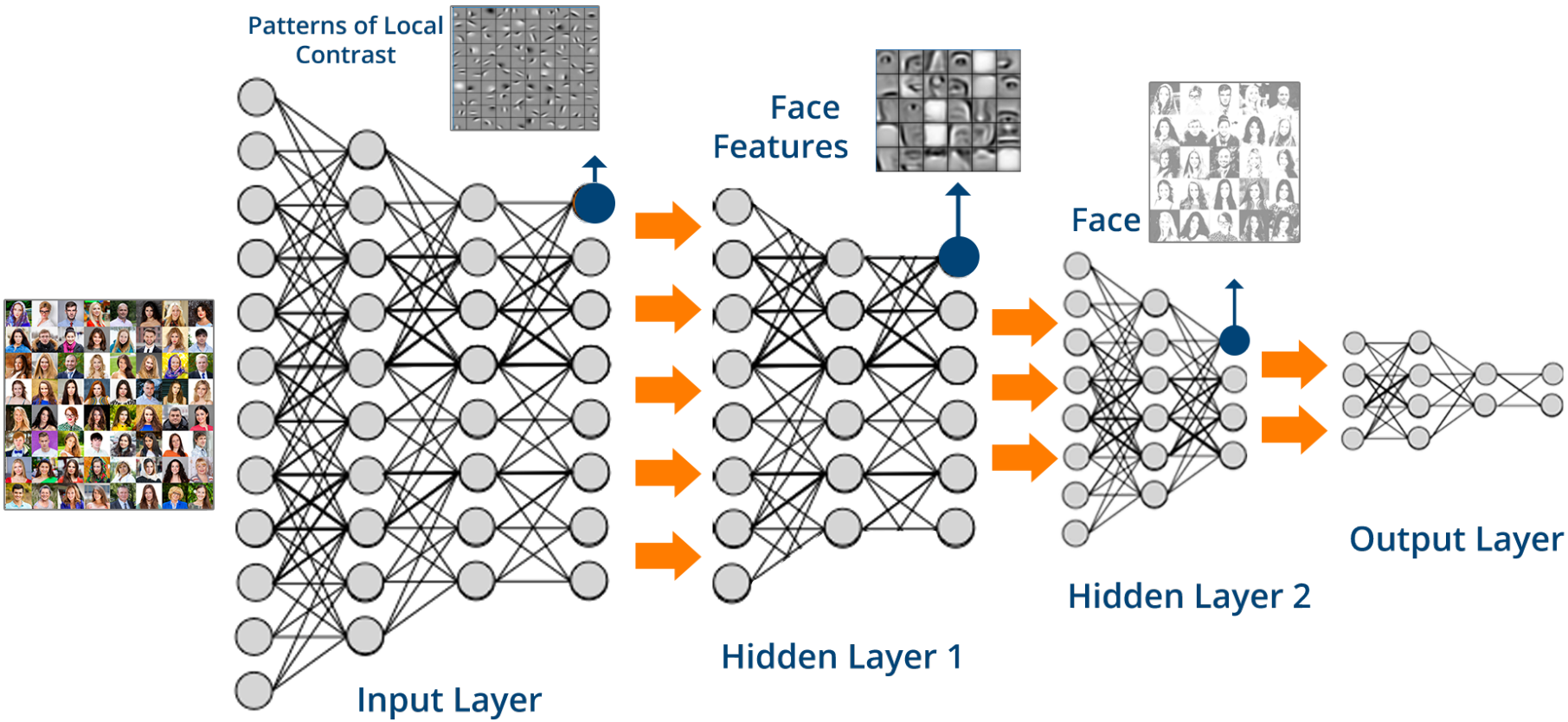


Machine Learning VS. Deep Learning

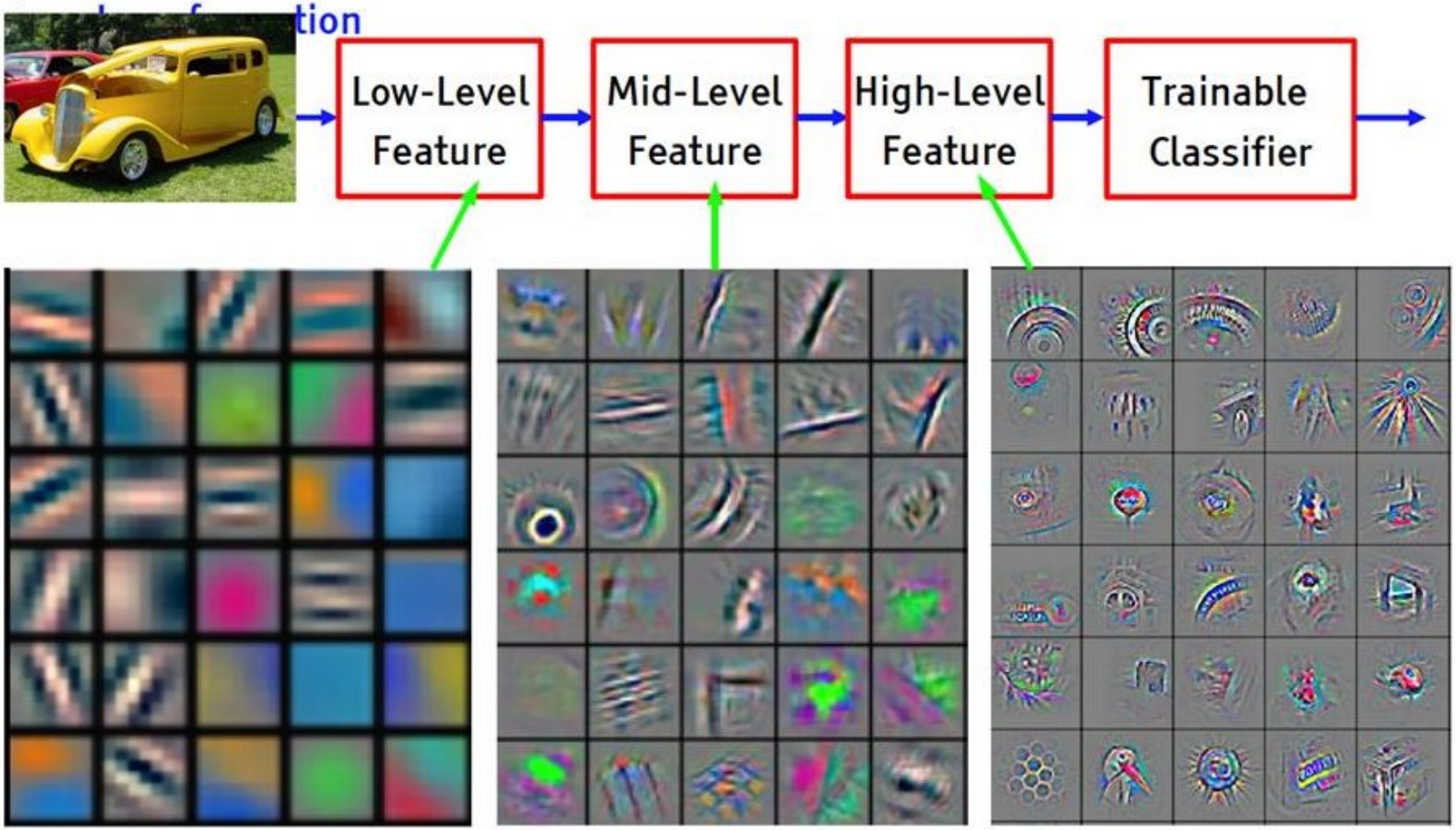




Deep Learning Learns Hierarchical Features



Deep Learning = Learning Hierarchical Representations



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

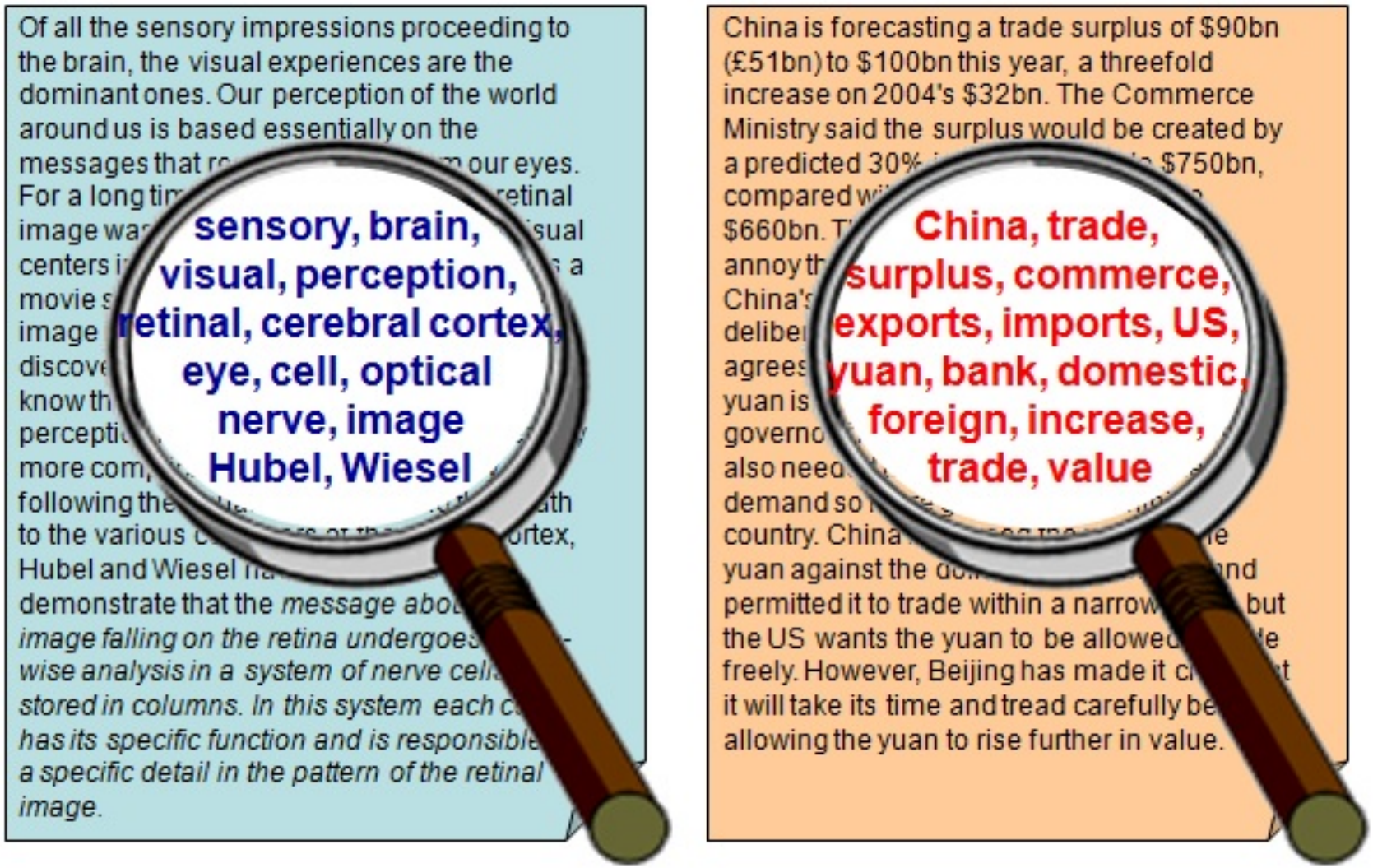
1. The Key Problem in AI: Feature Extraction

2. Visual Representation Before and After 2010
   - Bag-of-Visual-Words: The Landmark of Computer Vision in 2000-2010
     - Bag-of-Words Model for NLP / Text Mining
     - Bag-of-Visual-Words for Computer Vision
   - Deep Learning for Image Retrieval Since 2010
     - Autoencoders for Image Retrieval
     - Extracting Deep Features by ConvNets
   - Tips for Training ConvNets

3. Large Scale of Vehicles Search and Re-Identification

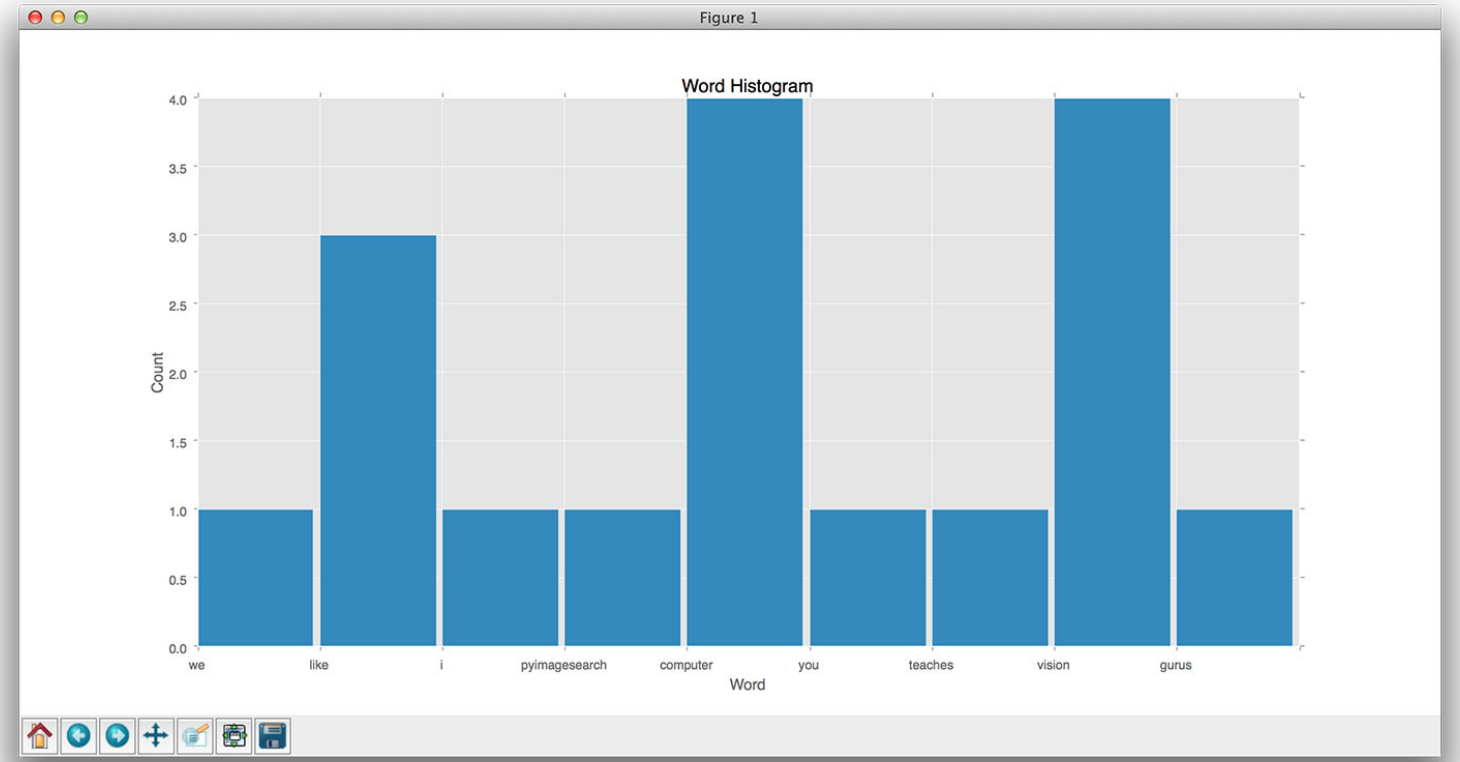
Document Representation: Bag-of-Words Model

A bag of words model entirely dicards the order of words in a document and simply counts the number of times each word appears.

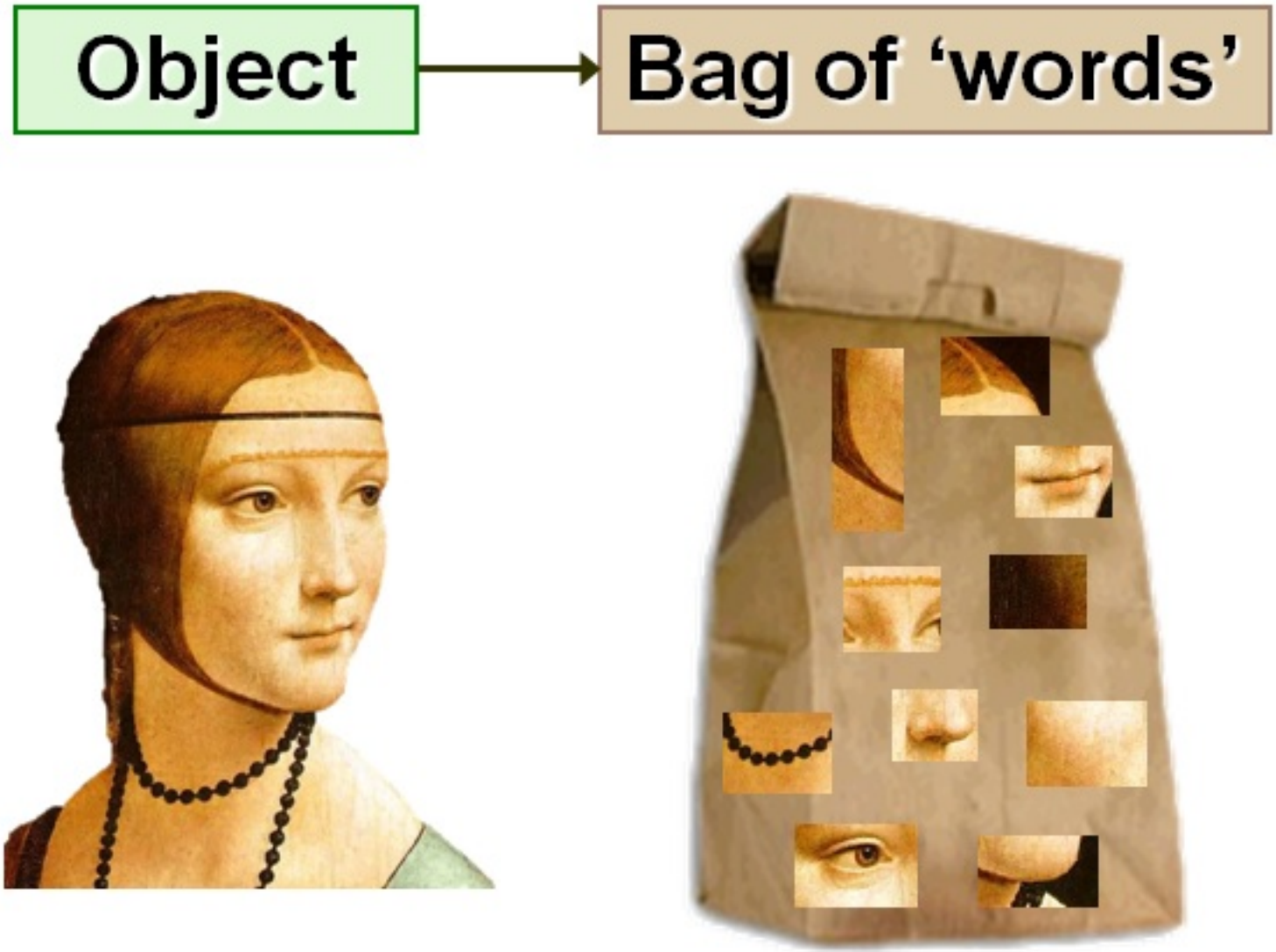


Bag-of-Words Example

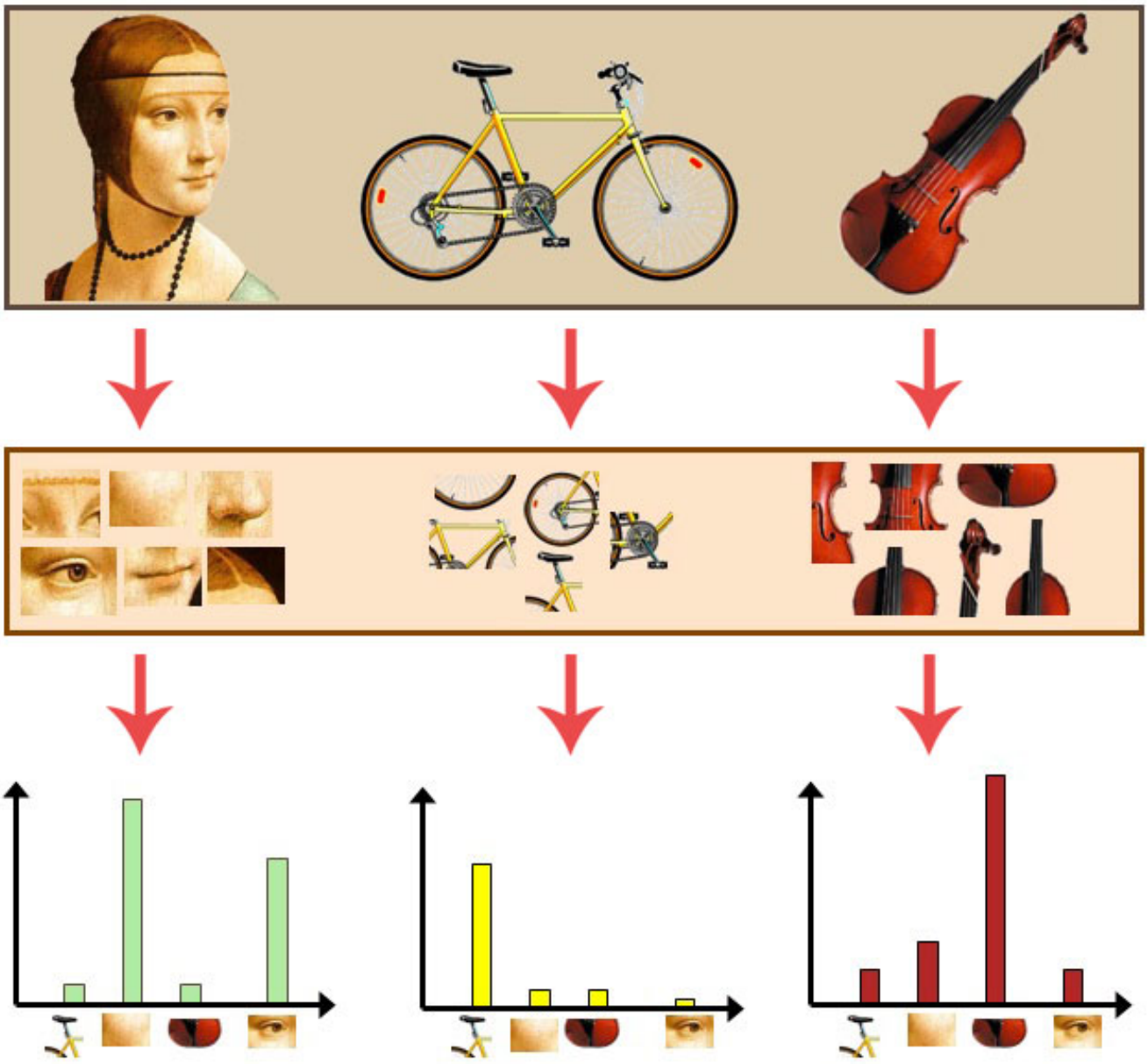
**"I like computer vision.
You like computer vision.
PyImageSearch Gurus
teaches computer vision."**



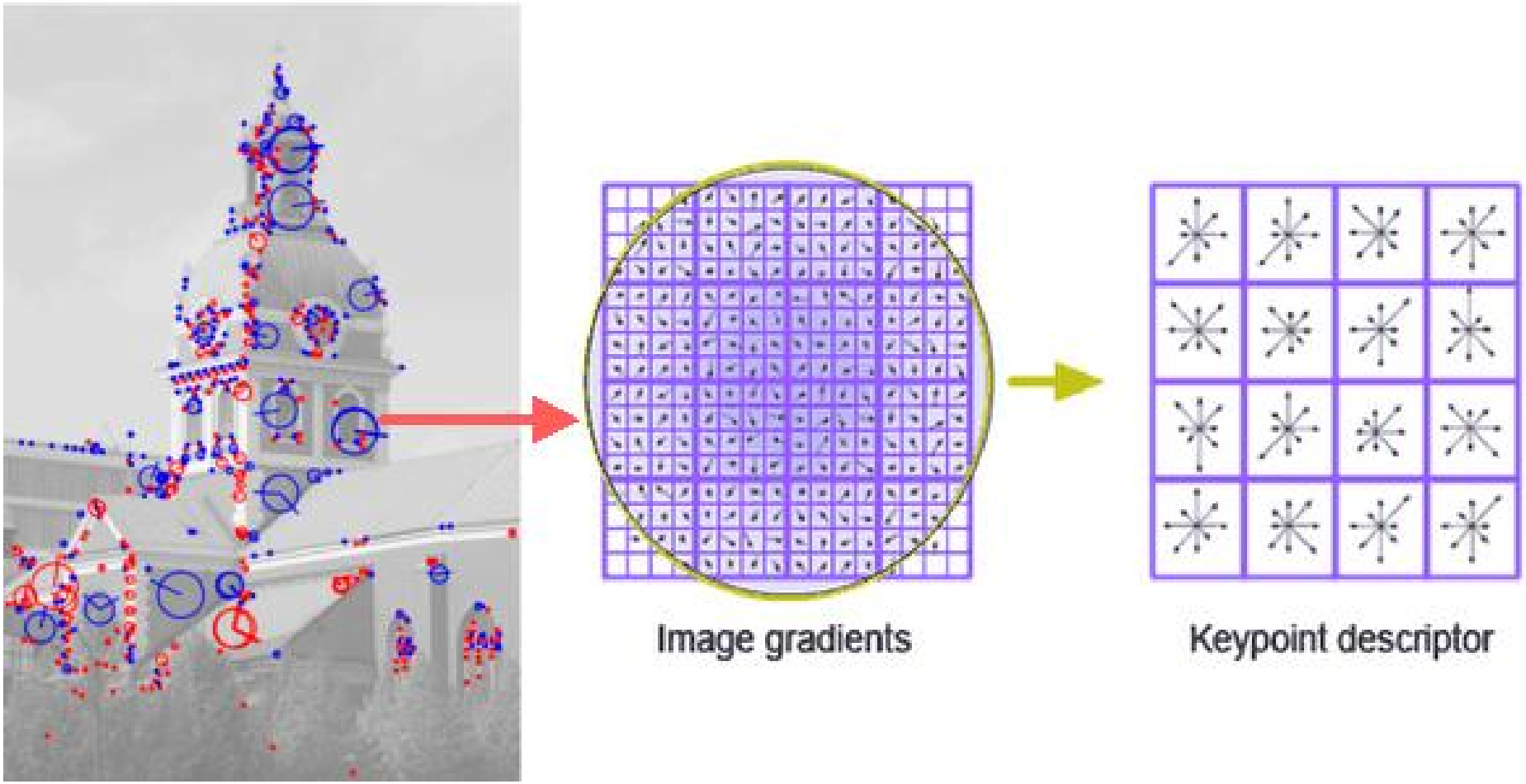
Visual Object Representation: Bag-of-Visual-Words



Bag-of-Visual-Words Example



The Visual Word: Local Feature



SIFT: Scale Invariant Feature Transform

Compute gradient orientation histsgrams of several small windows (to produce 128 values for each keypoint).

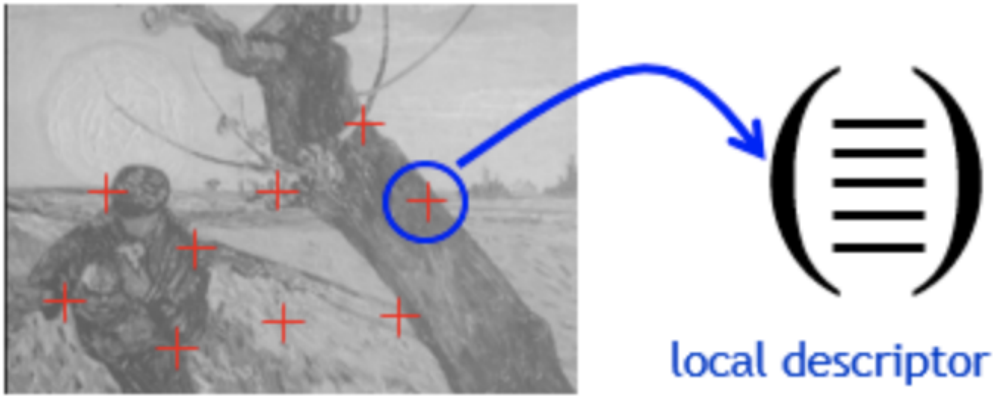
SIFT stages:

- 1.Scale-space extrema detection
- 2.Keypoint localization

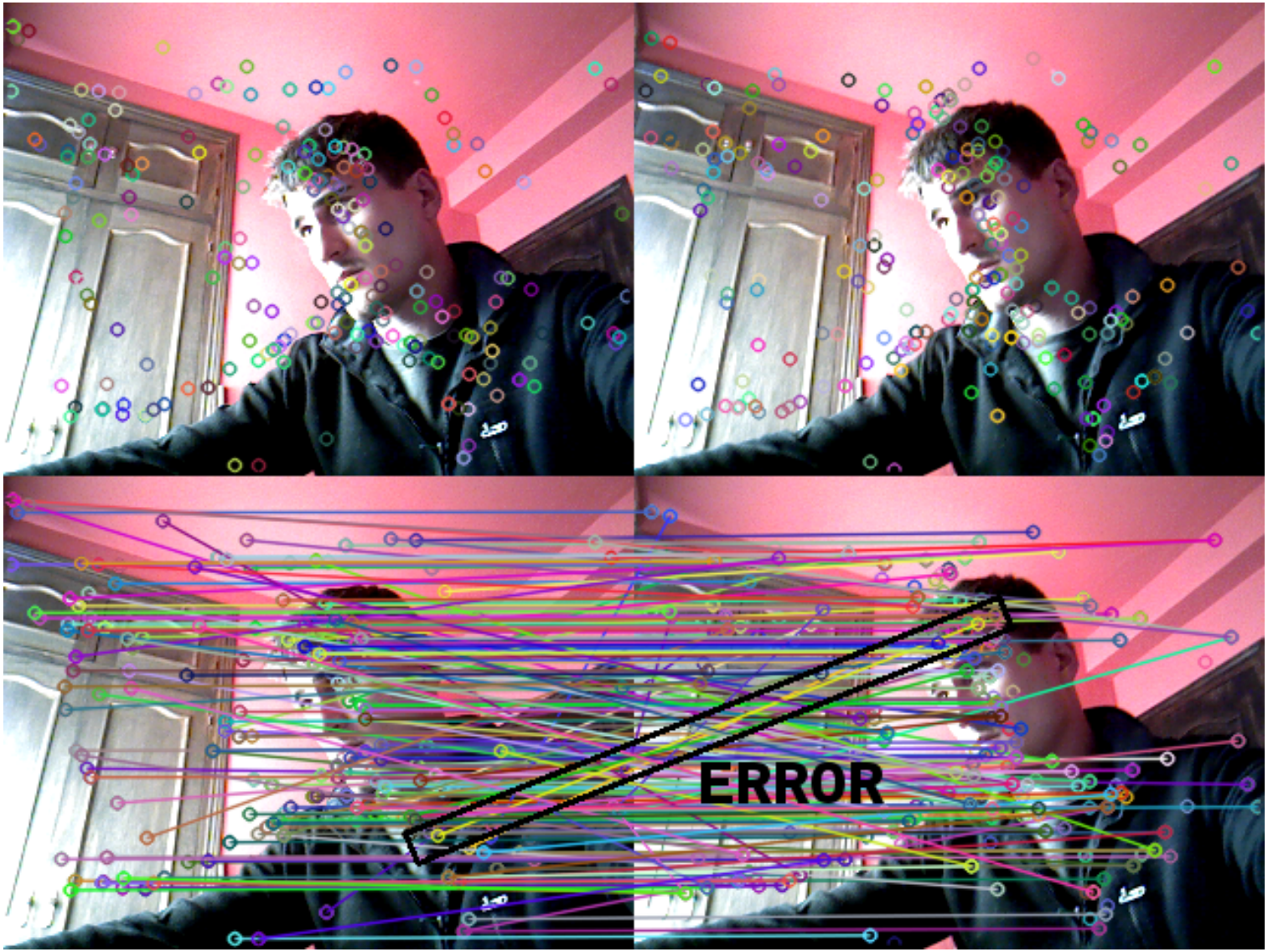
detector

- 3.Orientation assignment
- 4.Keypoint descriptor

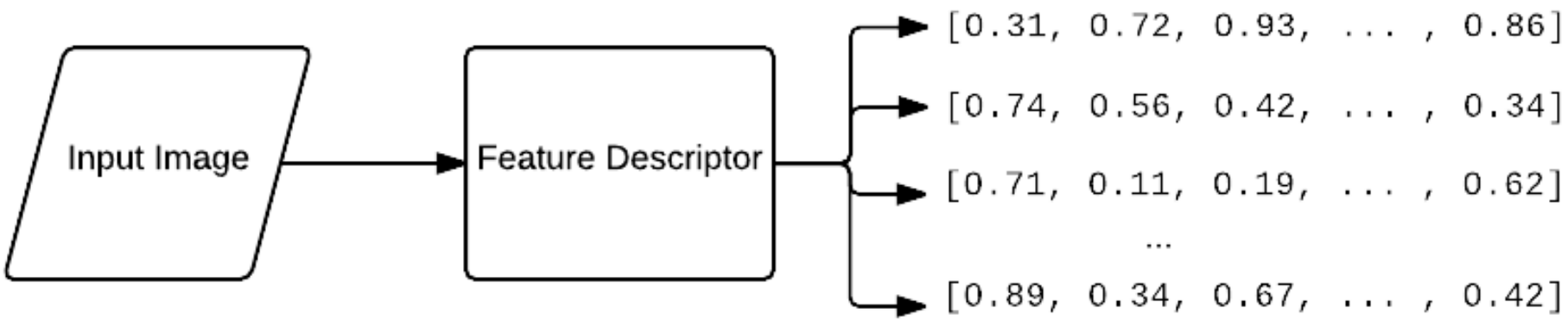
descriptor



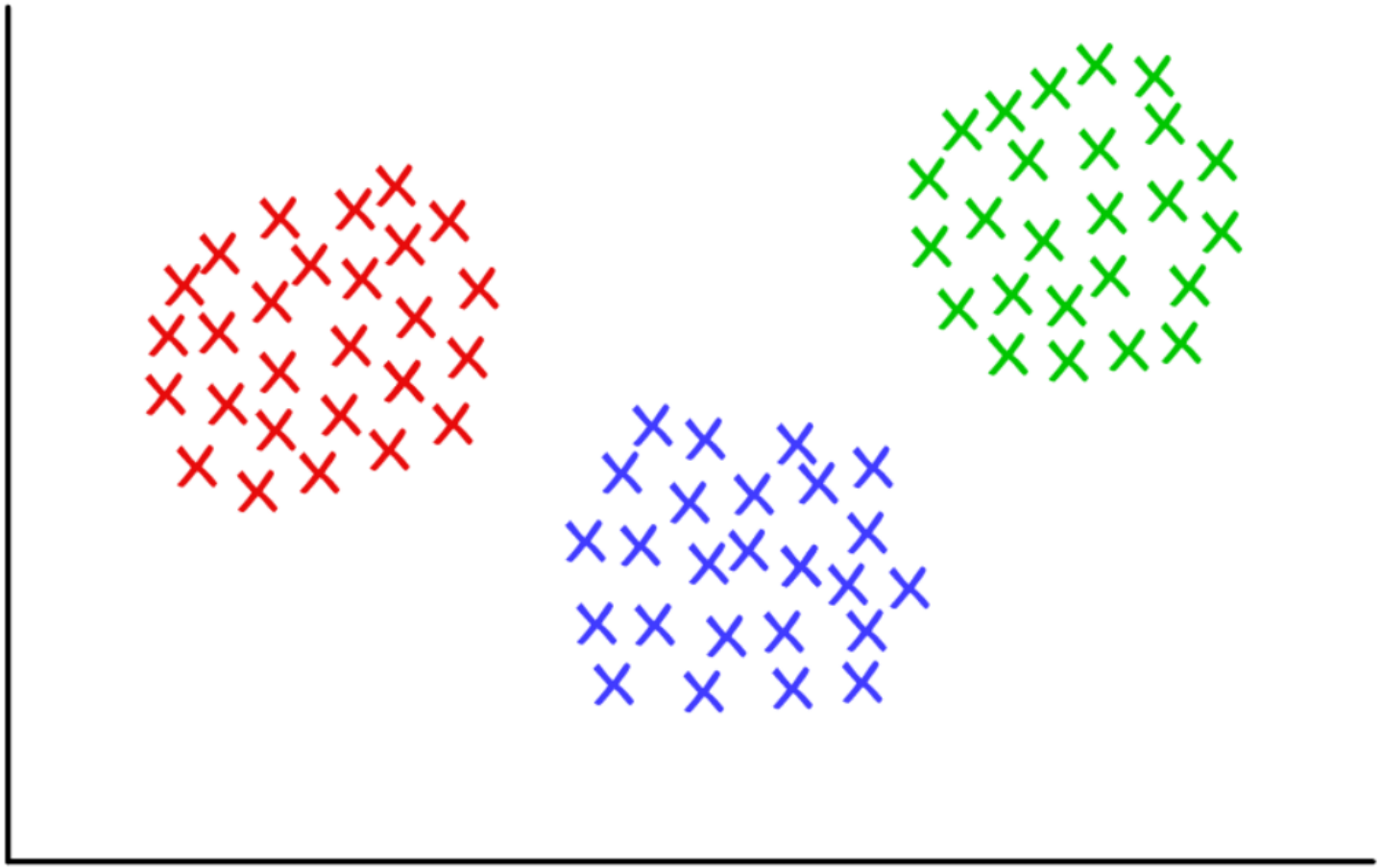
SIFT Matching



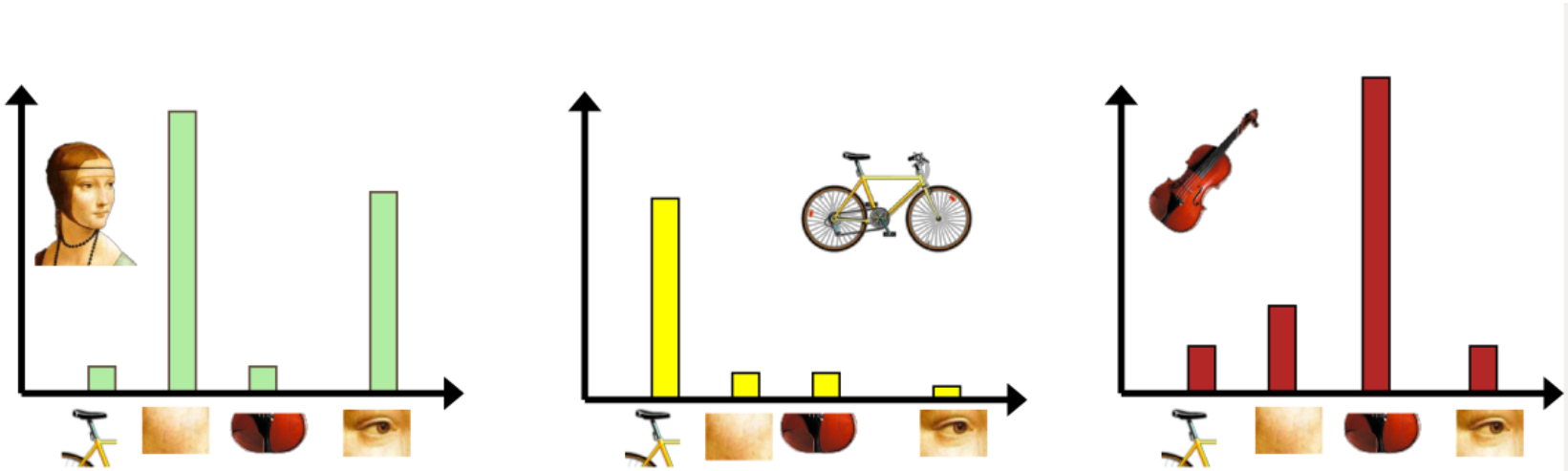
BoVW Step 1: Feature Extraction



BoVW Step 2: Codebook Construction

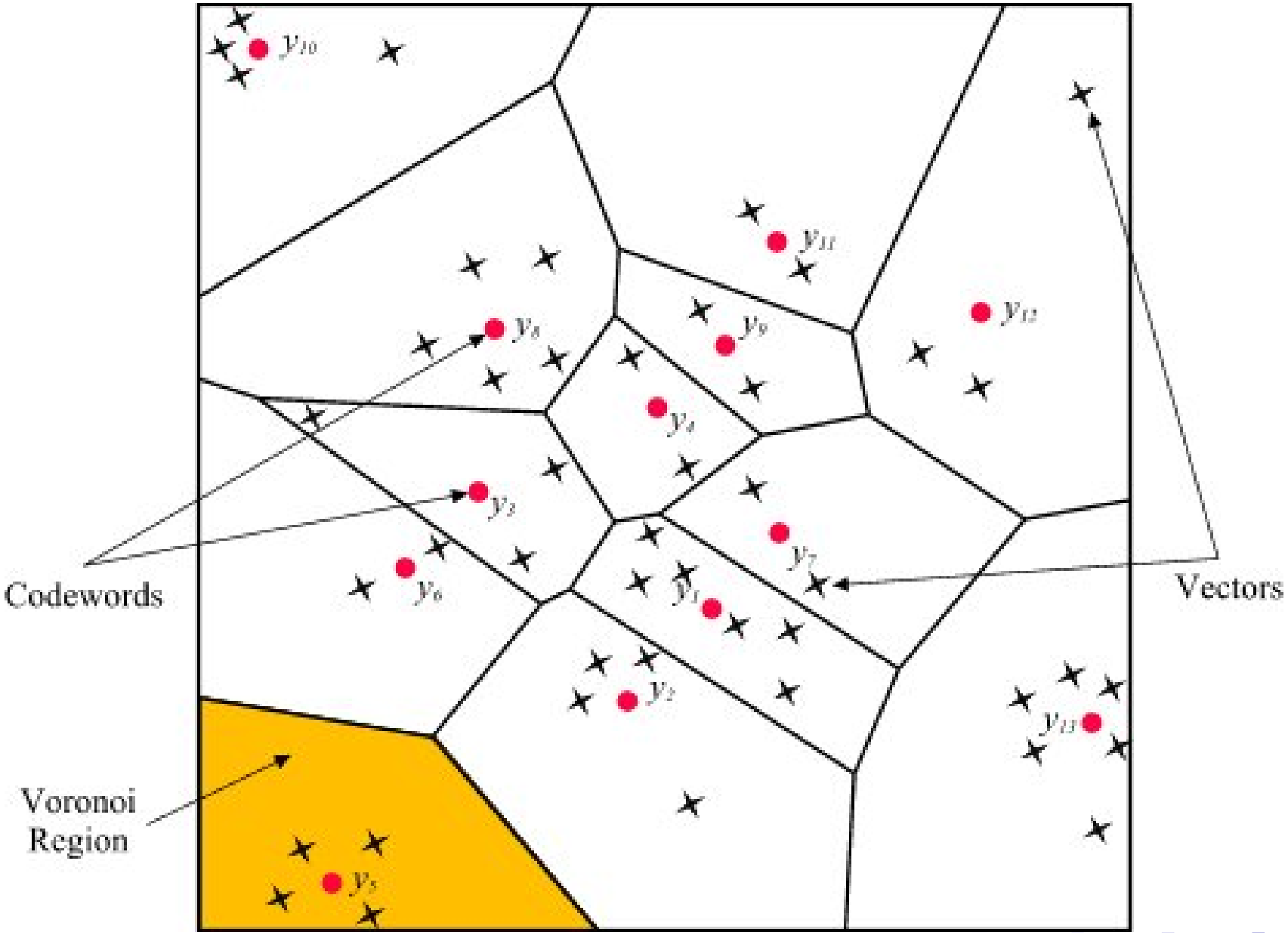


BoVW Step 2: Codebook Construction

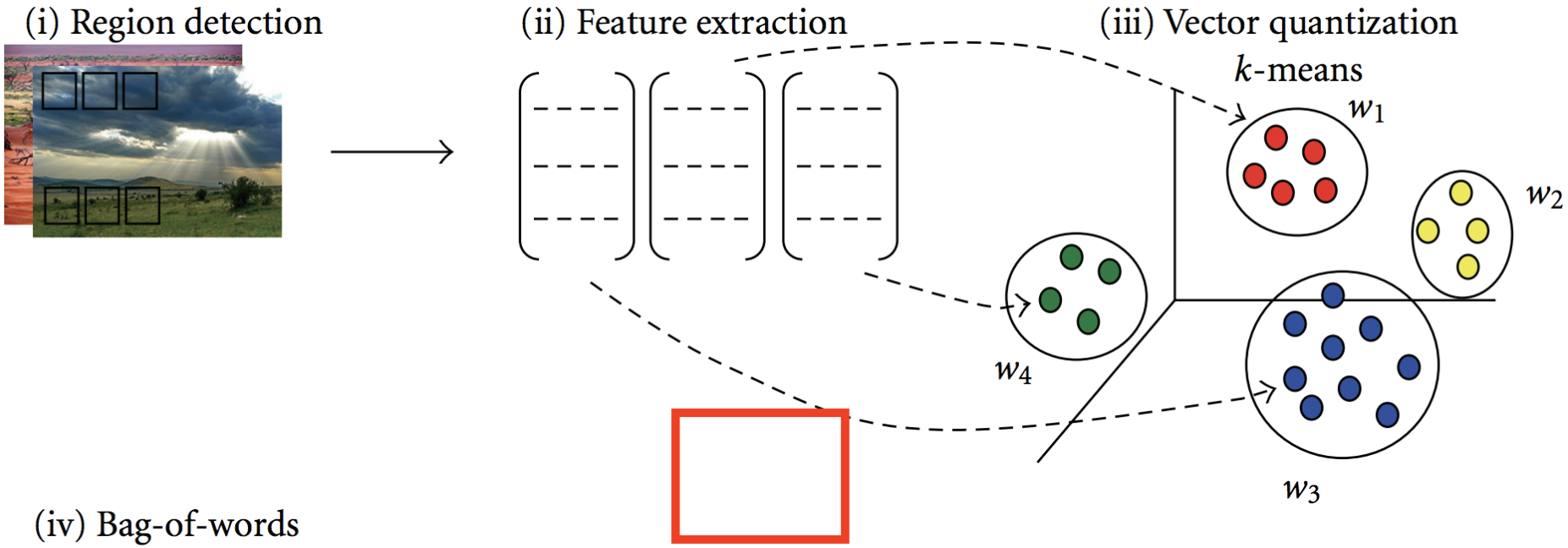


codewords dictionary

BoVW Step 3: Vector Quantization

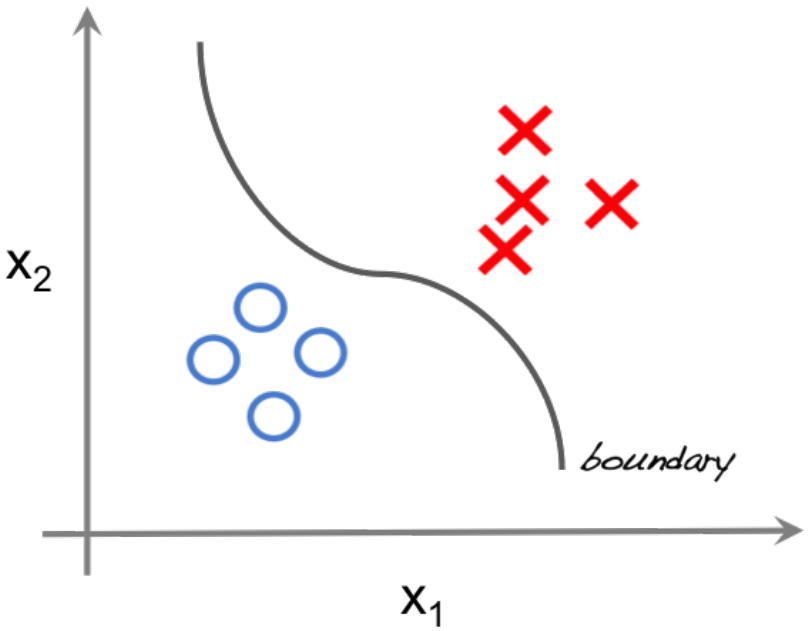


BoVW Workflow

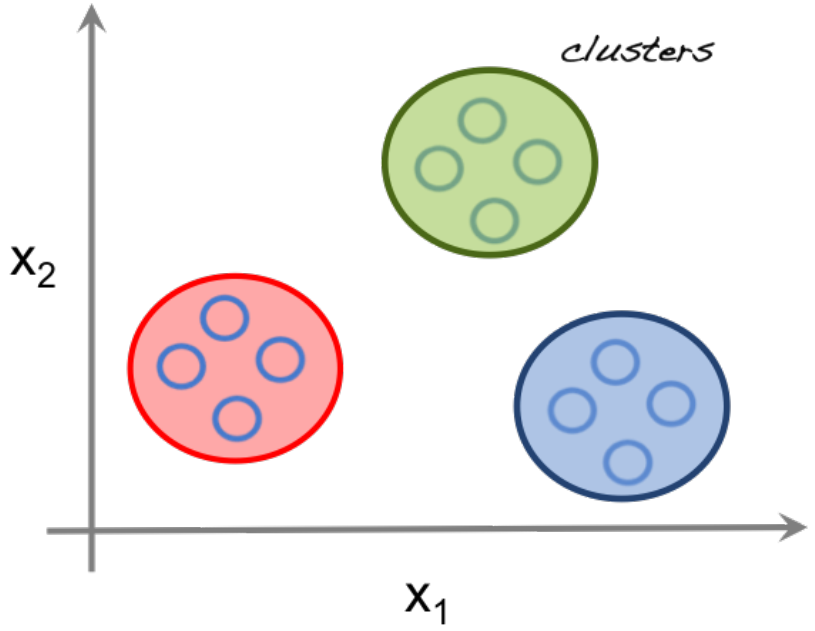


Supervised Learning VS. Unsupervised Learning

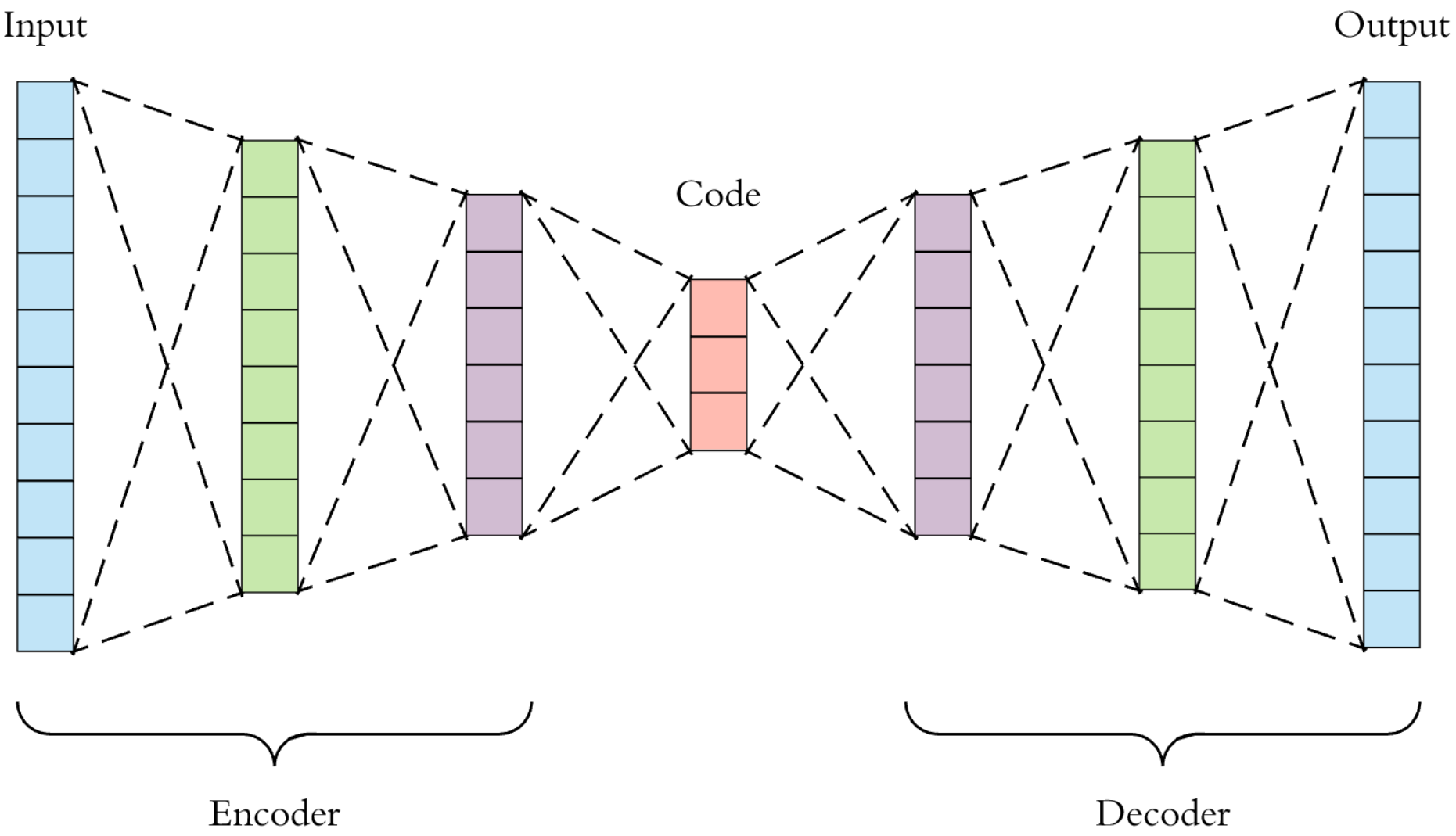
Supervised learning



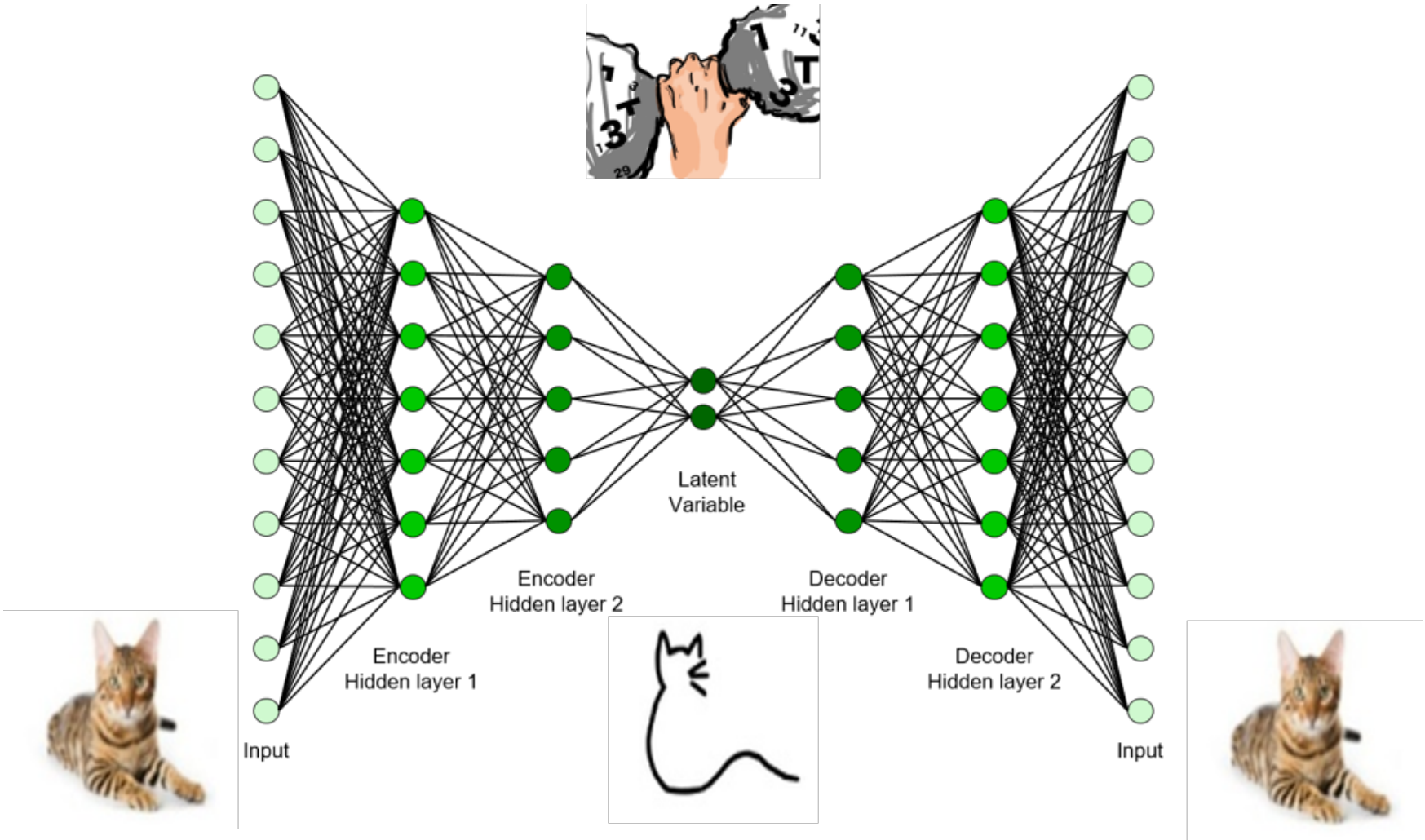
Unsupervised learning



Encoder-Decoder Framework

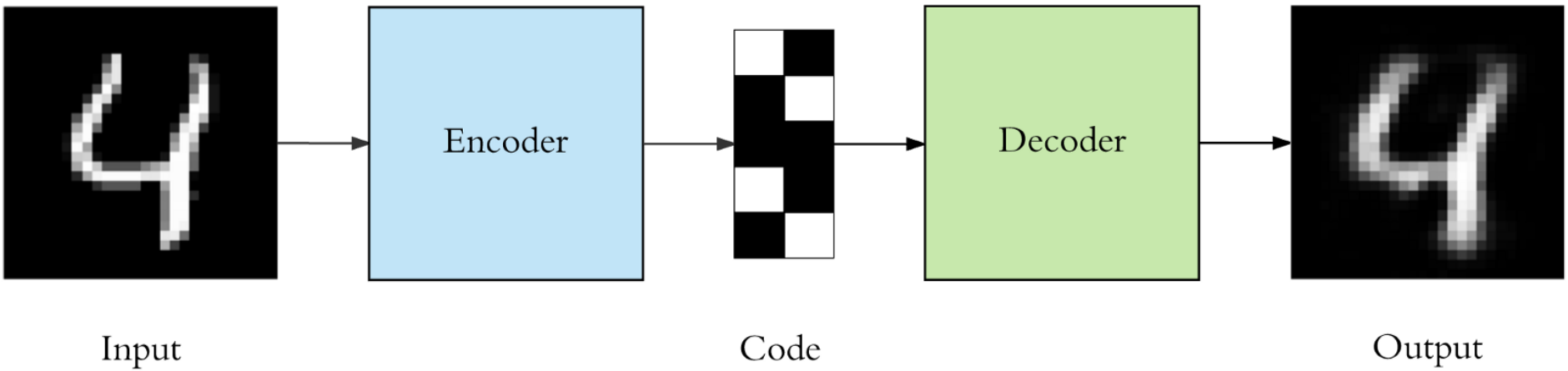


Autoencoder



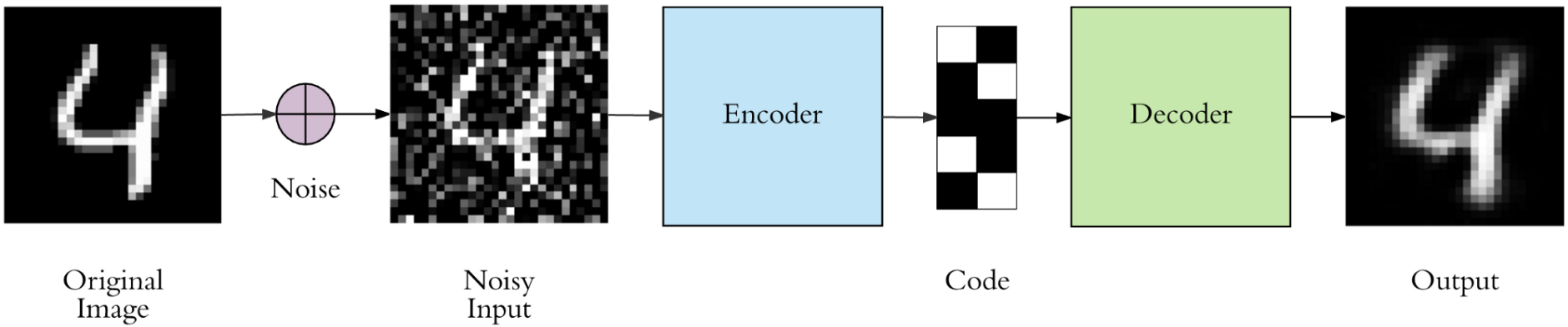
Advantages of Autoencoder

- Unsupervised learning which does not require labels.
- Feature representation in the middle of the encoder-decoder framework.
- Useful for data compression.



Denoising Autoencoder

Denoising autoencoder adds random noise to its inputs and makes it recover the original noise-free data. This way the autoencoder can not simply copy the input to its output because the input also contains random noise.

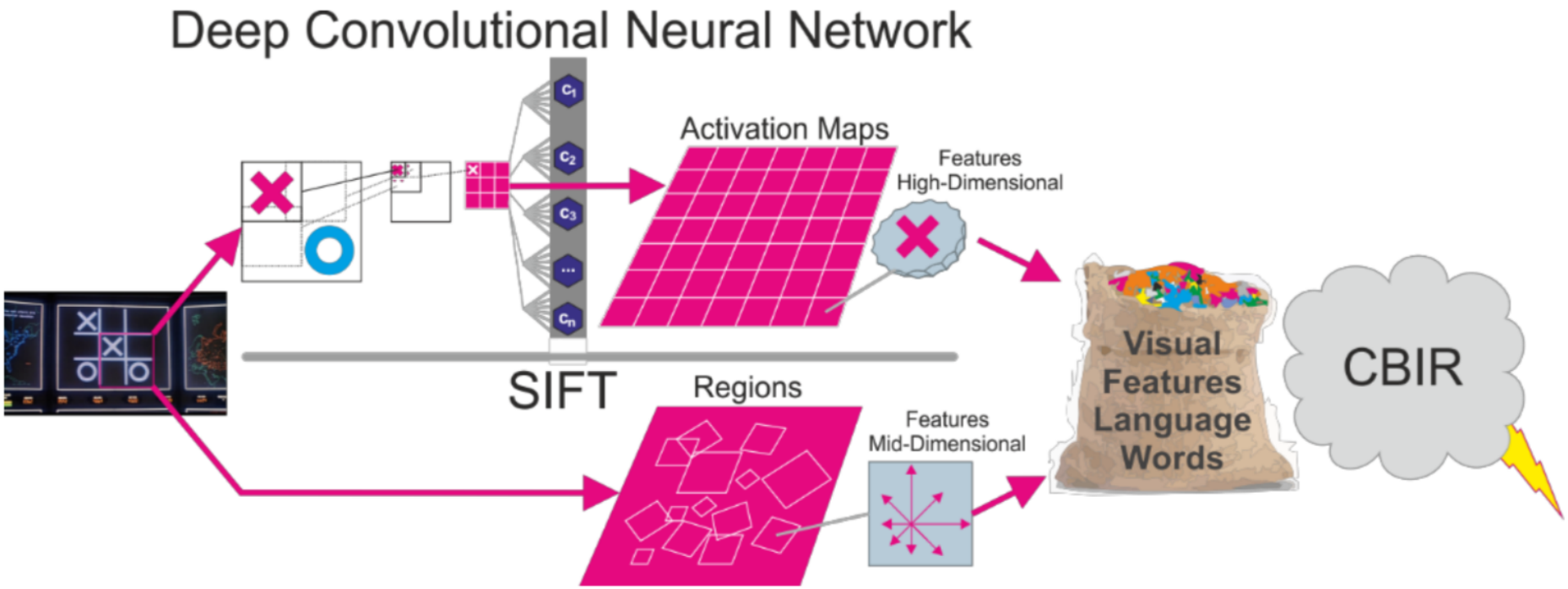


Denosing Autoencoders on MNIST

Query image

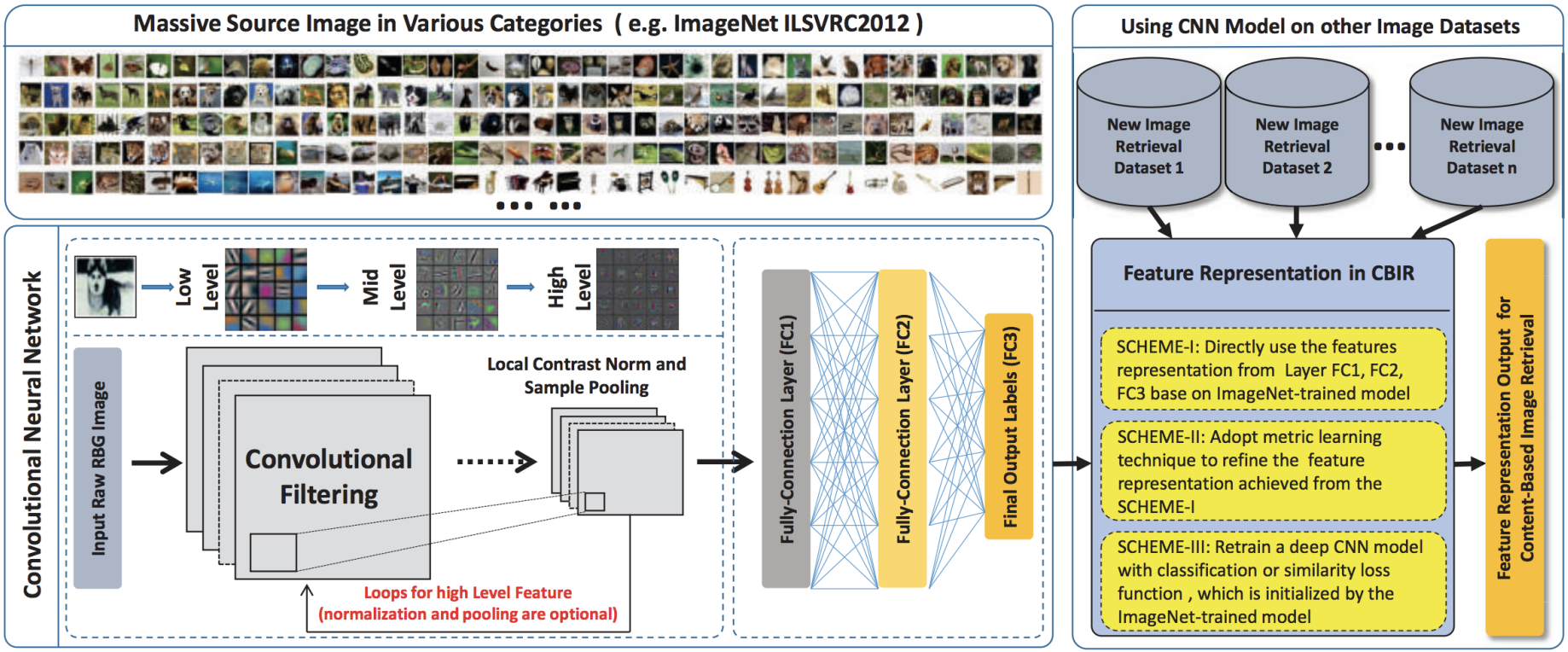
First 10 retrieved images

BovW VS. ConvNets



Deep Learning for CBIR

Deep Learning for Content-Based Image Retrieval: A Comprehensive Study.



(a) Training deep CNN models in an existing domain (ImageNet)

(b) Adopting trained model for CBIR in a new domain

Best Practice For CNNs Applied to Visual IR

- Feature aggregation and normalization.
- Output layer selection.
- Image resizing.
- Multi-scale feature representation.
- PCA and whitening.

What Is the Best Practice for CNNs Applied to Visual Instance Retrieval?

Jiedong Hao, Jing Dong, Wei Wang, Tieniu Tan
Center for Research on Intelligent Perception and Computing
Institute of Automation, Chinese Academy of Sciences

Abstract

Previous work has shown that feature maps of deep convolutional neural networks (CNNs) can be interpreted as feature representation of a particular image region. Features aggregated from these feature maps have been exploited for image retrieval tasks and achieved state-of-the-art performances in recent years. The key to the success of such methods is the feature representation. However, the different factors that impact the effectiveness of features are still not explored thoroughly. There are much less discussion about the best combination of them.

The main contribution of our paper is the thorough evaluations of the various factors that affect the discriminative ability of the features extracted from CNNs. Based on the evaluation results, we also identify the best choices for different factors and propose a new multi-scale image feature representation method to encode the image effectively. Finally, we show that the proposed method generalises well and outperforms the state-of-the-art methods on four typical datasets used for visual instance retrieval.

Feature Aggregation and Normalization

Max-pooling, sum-pooling, L_1 normalization, L_2 normalization.

Table 1: Comparison between different combinations of feature aggregation and normalization methods.

Method	full-query	cropped-query
max-l_1	52.4	48.0
sum-l_2	58.0	52.6
sum-l_1	60.3	56.3
max-l_2	60.1	53.5

Output Layer Selection

Choose Vgg-19 network and change the fully-connected layers to convolutional layers.

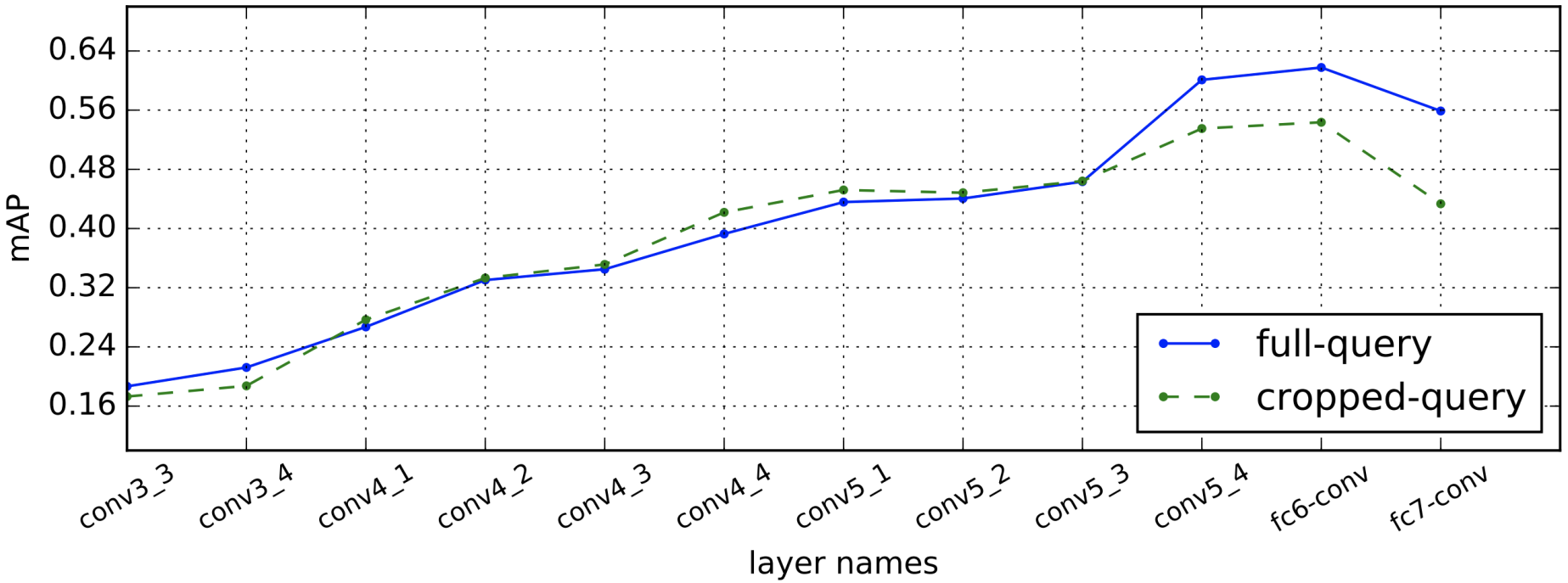


Image Resizing

- Both the height and width of the dataset images are set to the same fixed value. (*two-fixed*)
- The minimum of each dataset image's size is set to a fixed value. The aspect ratio of the original image is kept. (*one-fixed*)
- The images are kept their original sizes. (*free*)

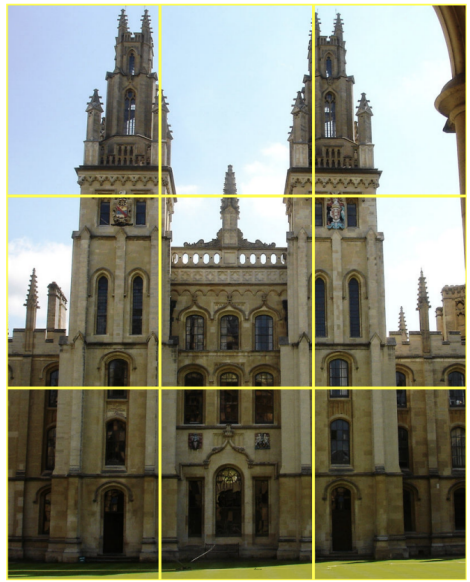
Table 2: **Comparison between different image resizing strategies.** The numbers in the parentheses denote the sizes in which the maximum mAPs are achieved.

Method	full-query	cropped-query
two-fixed	55.5 (864)	38.7 (896)
one-fixed	59.0 (800)	39.3 (737)
free	58.0	52.6

Multi-scale Feature Representation

(a) level 1

(b) level 2

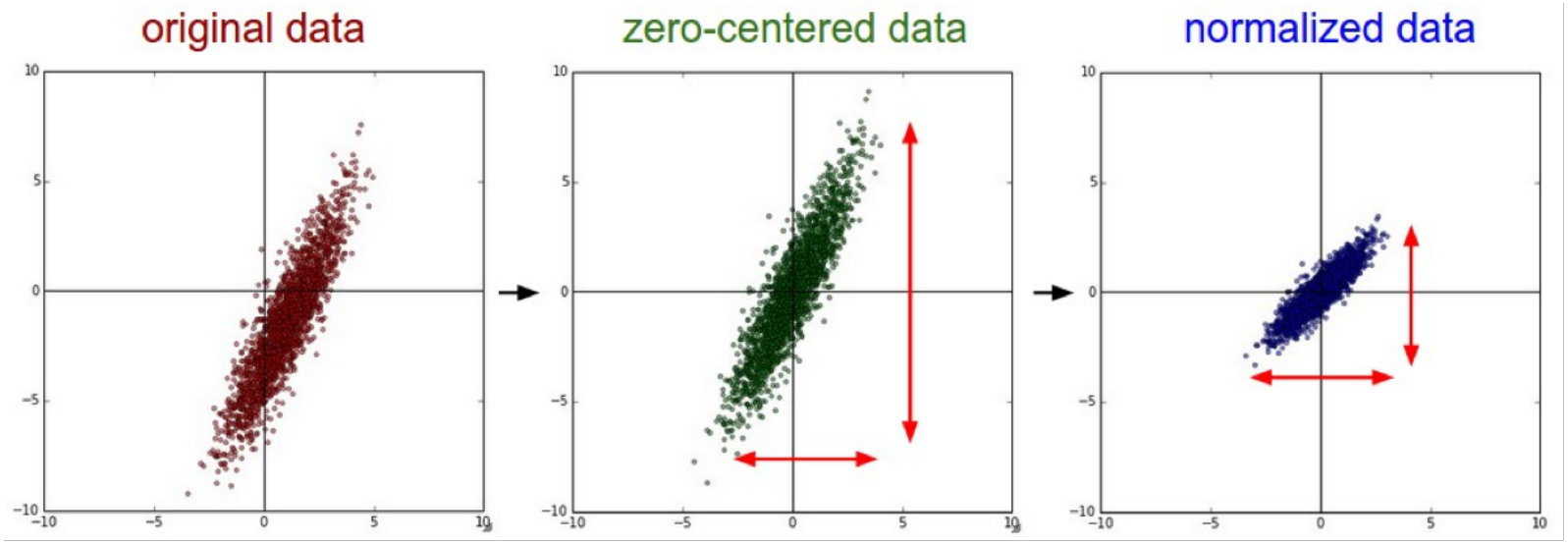


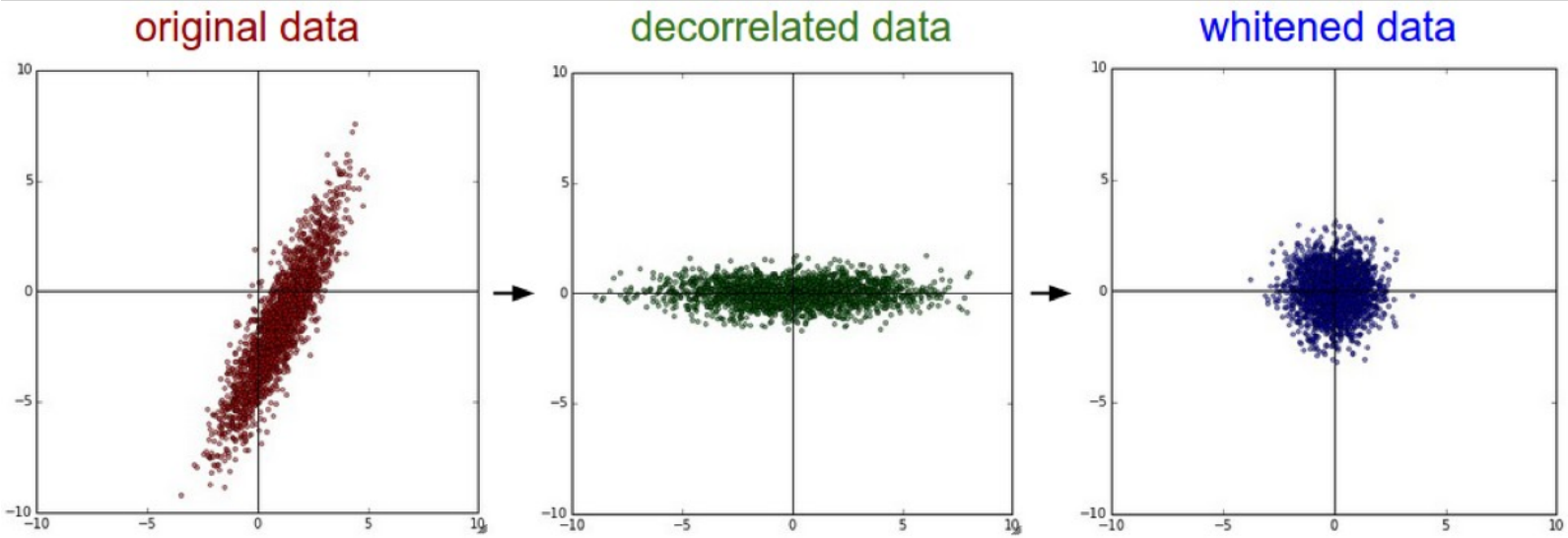
(c) level 3

Figure 1: **An illustration of multi-scale representation of an image.** The whole image is divided into 3 levels from the coarsest (level 1) to the finest (level 3). At each level, the image is divided into different number of equal-sized regions.

PCA and Whitening

The motivation of PCA and whitening is to decorrelate and normalize data.



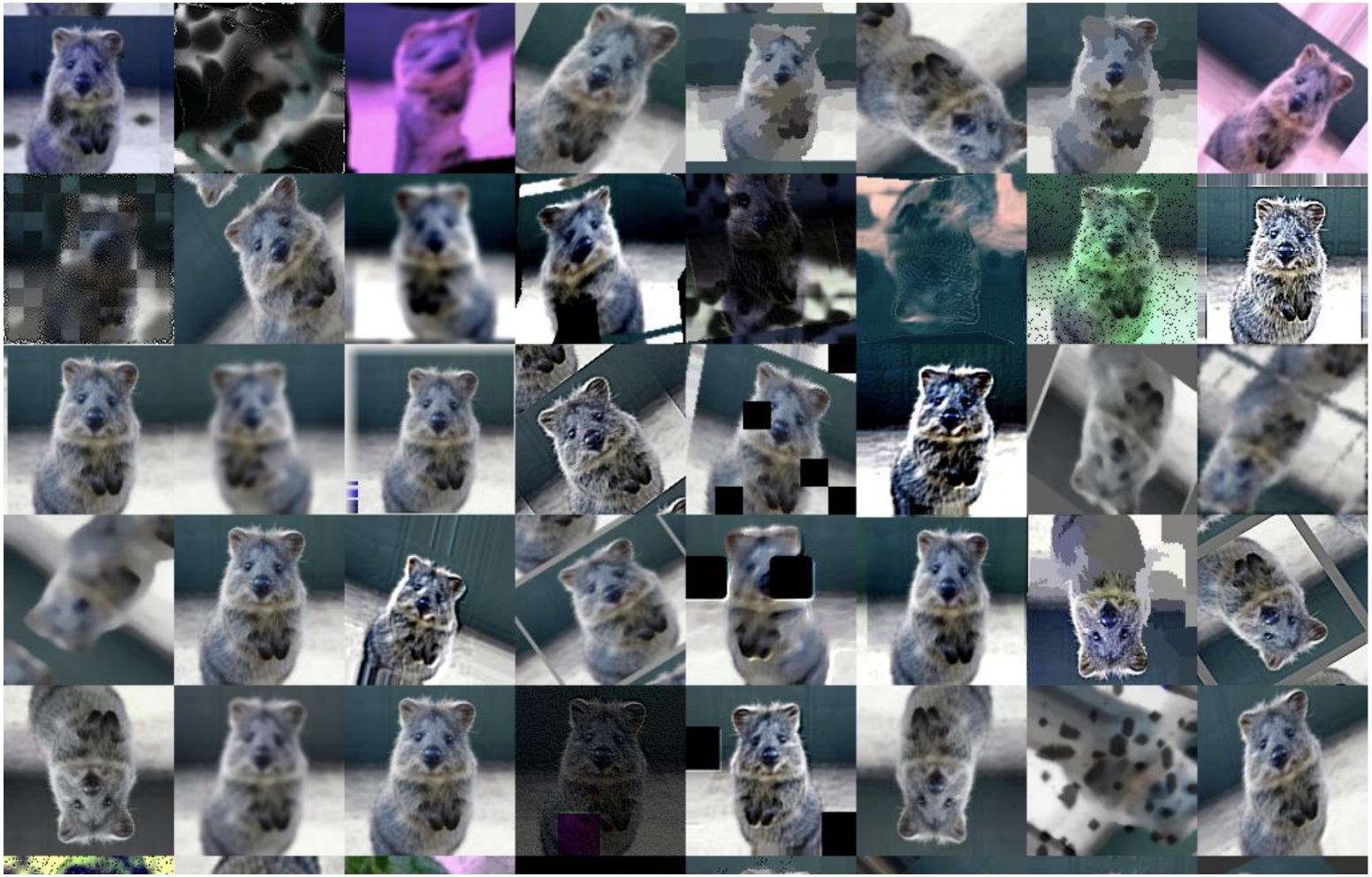


Training ConvNets Tips

- Data augmentation.
- Pre-Processing. (1) Zero-center the data and then normalize them; (2) PCA whitening.
- Initializations. Kaiming He initialization: $w = np.random.randn(n) * sqrt(2.0/n)$
- Decay learning rate during training process.
- Fine-tune on pre-trained models.
- Activation functions. Use ReLU or Leaky ReLU instead of Sigmoid function.
- Regularizations. L_1 regularization, L_2 regularization, or Dropout.
- Reading insights from traing accuracy or loss figures.
- Ensemble. (1) Same model, different initialization; (2) Top models discovered during cross-validation; (3) Different checkpoints of a single model.
- Be careful with class-imbalanced data. Use weighted loss or batch-wise balanced sampling.

Data Augmentation Lib

https://github.com/aleju/imgaug



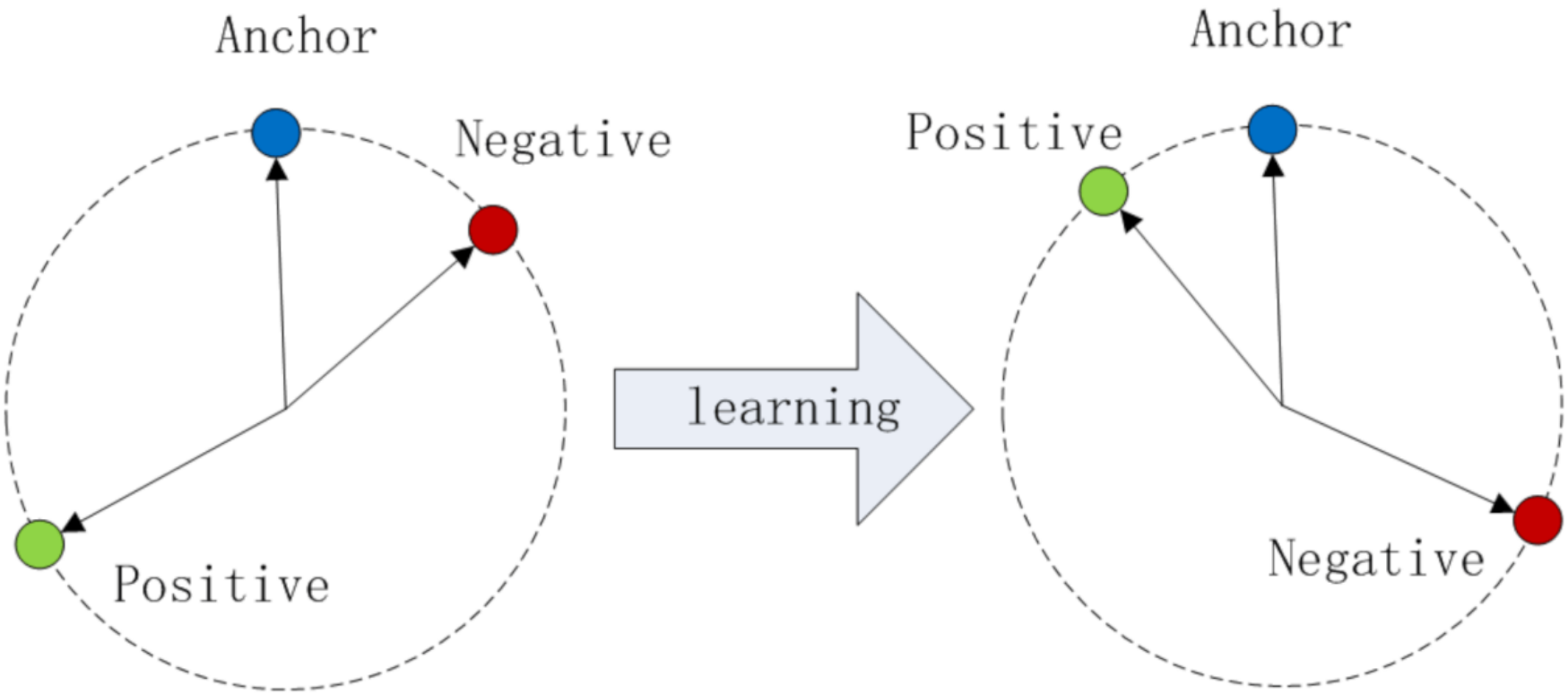
1. The Key Problem in AI: Feature Extraction

2. Visual Representation Before and After 2010
   - Bag-of-Words Model for NLP / Text Mining
   - Bag-of-Visual-Words for Computer Vision
   - Autoencoders for Image Retrieval
   - Extracting Deep Features by ConvNets

3. Large Scale of Vehicles Search and Re-Identification
   - Triplet Loss
   - Apply Triplet Loss Network to Solve Vehicle Re-Identification

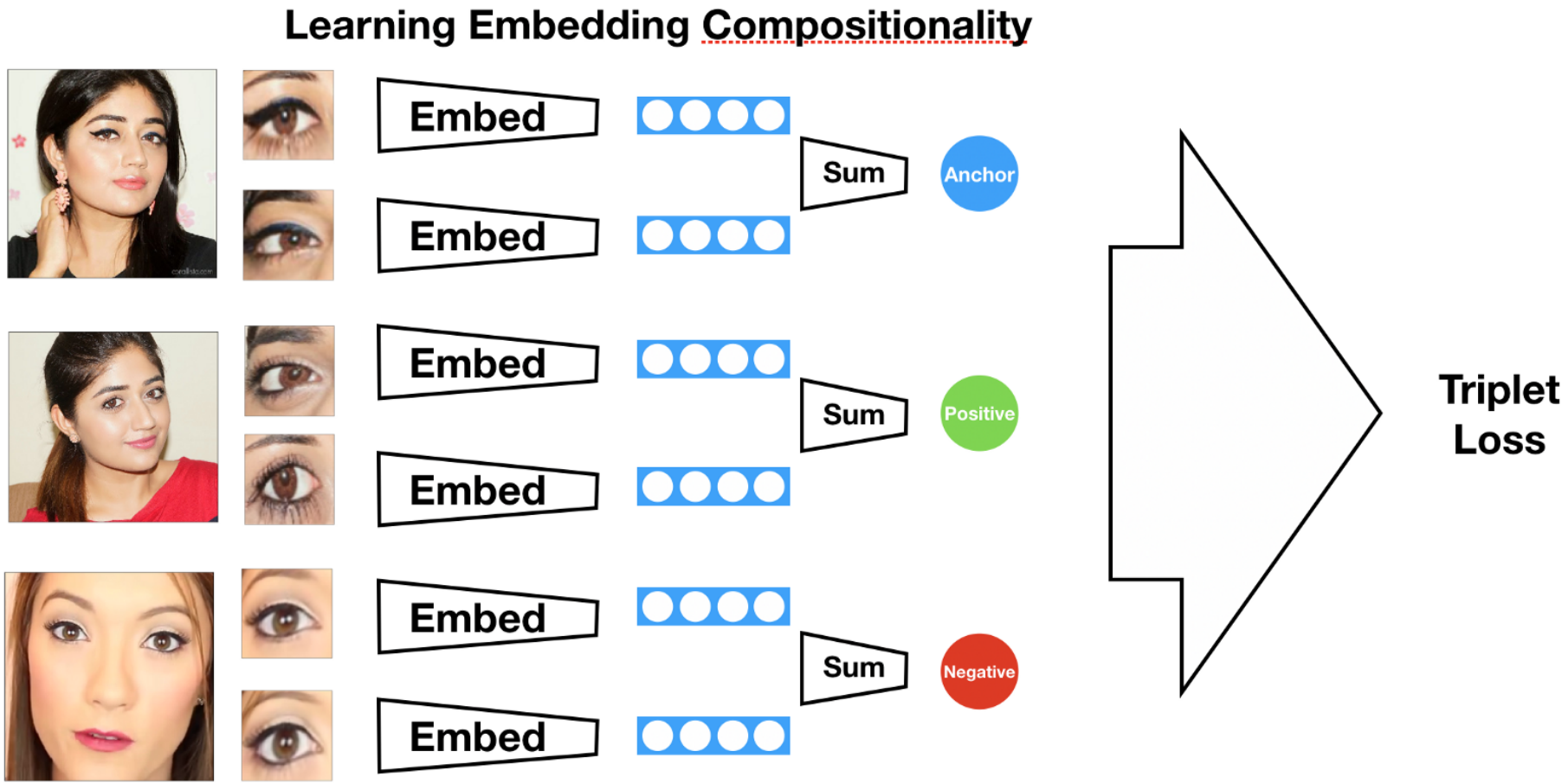
Triplet Loss

Triplet loss tries to learn a feature embedding, which ensures that the distance between the anchor and the positive is smaller than the distance between the anchor and the negative.



Triplet Loss in Face Recognition

FaceNet: A Unified Embedding for Face Recognition and Clustering.



Triplet Loss Derivation

$$Loss = \sum_{i=1}^{N} \left[\| f_i^a - f_i^p \|_2^2 - \| f_i^a - f_i^n \|_2^2 + \alpha \right]_+$$

The gradient w.r.t the "anchor" input (`fa`):

$$\frac{\partial Loss}{\partial f^a} = \sum_{i=1}^{N} \begin{cases} 2\left(f_i^n - f_i^p\right) & \text{if } \left(\| f_i^a - f_i^p \|_2^2 - \| f_i^a - f_i^n \|_2^2 + \alpha \right) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

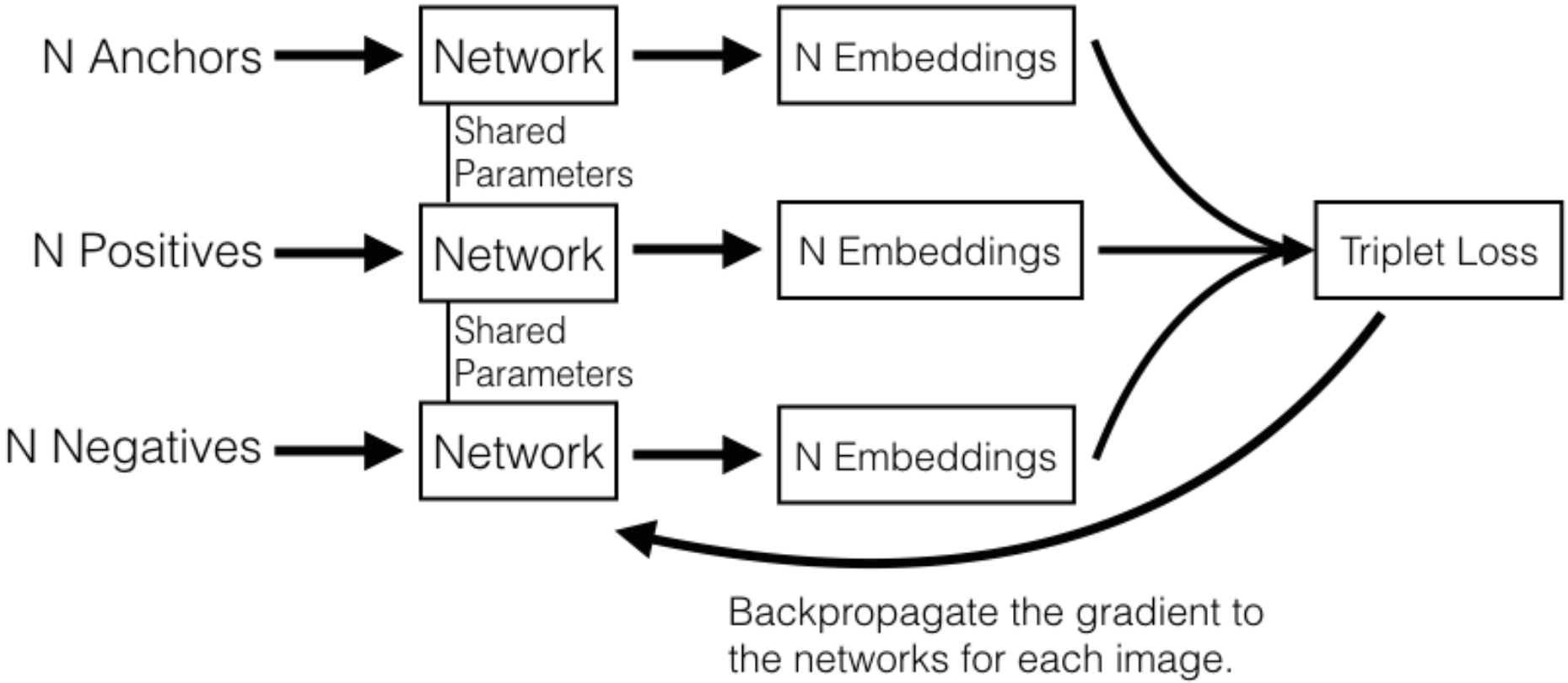
The gradient w.r.t the "positive" input (`fp`):

$$\frac{\partial Loss}{\partial f^p} = \sum_{i=1}^{N} \begin{cases} -2\left(f_i^a - f_i^p\right) & \text{if } \left(\| f_i^a - f_i^p \|_2^2 - \| f_i^a - f_i^n \|_2^2 + \alpha \right) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The gradient w.r.t the "negative" input (`fn`):

$$\frac{\partial Loss}{\partial f^n} = \sum_{i=1}^{N} \begin{cases} 2\left(f_i^a - f_i^n\right) & \text{if } \left(\| f_i^a - f_i^p \|_2^2 - \| f_i^a - f_i^n \|_2^2 + \alpha \right) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Triplet Loss Network Architecture



1. The Key Problem in AI: Feature Extraction

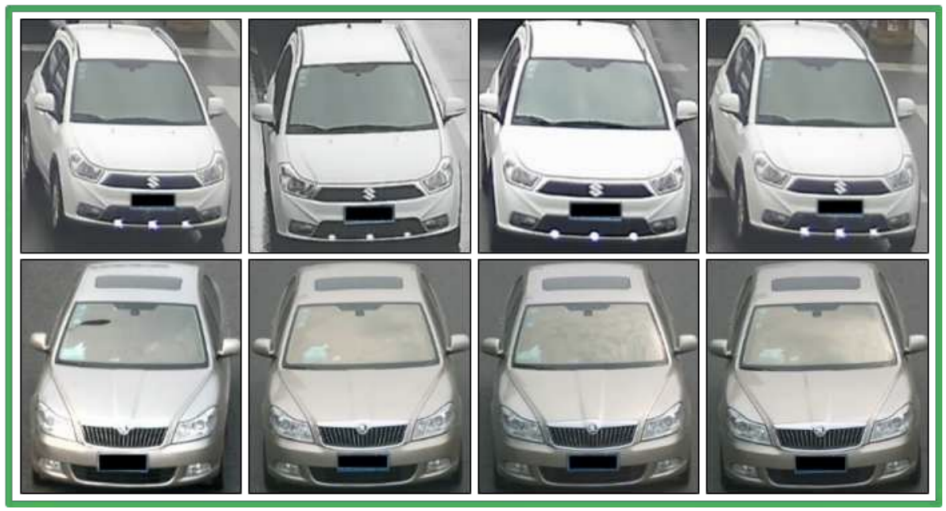
2. Visual Representation Before and After 2010
   - Bag-of-Words Model for NLP / Text Mining
   - Bag-of-Visual-Words for Computer Vision
   - Autoencoders for Image Retrieval
   - Extracting Deep Features by ConvNets

3. Large Scale of Vehicles Search and Re-Identification
   - Triplet Loss
   - Apply Triplet Loss Network to Solve Vehicle Re-Identification

Triplet Loss Network Application: Vehicle Re-Identificaiton

Different cars

Same cars



VehicleID: Vehicle Re-Identificaiton Benchmark

20W+ Images, 3W+ Vehicles. Each image is associated with two attributes: model and color.

Method 1: Train ConvNets to Predict Vehicle Models

http://www.eecs.qmul.ac.uk/ xiatian/papers/Kanac

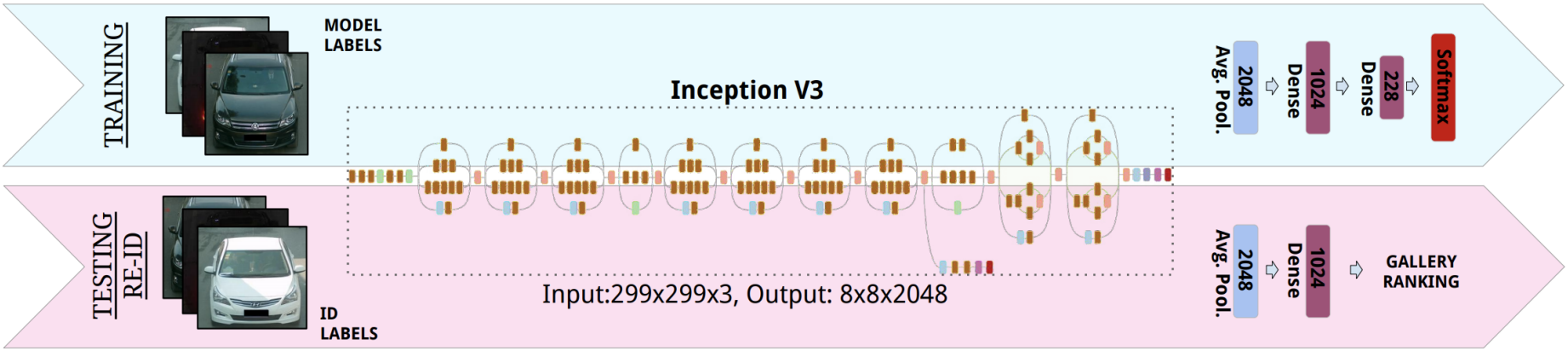


Figure 2: Overview of the proposed Cross-Level Vehicle Recognition (CLVR) method for vehicle re-identification: **(1) Training** (vehicle model classification): Learn a less fine-grained vehicle model classification deep model by a customised Inception-V3 [26] CNN network; **(2) Deployment** (vehicle re-id matching): Deploy the learned CLVR model as a feature extractor using the output of the fully-connected feature (Dense-1024) layer for more fine-grained vehicle re-id tasks.

Method 1: Code Snippet

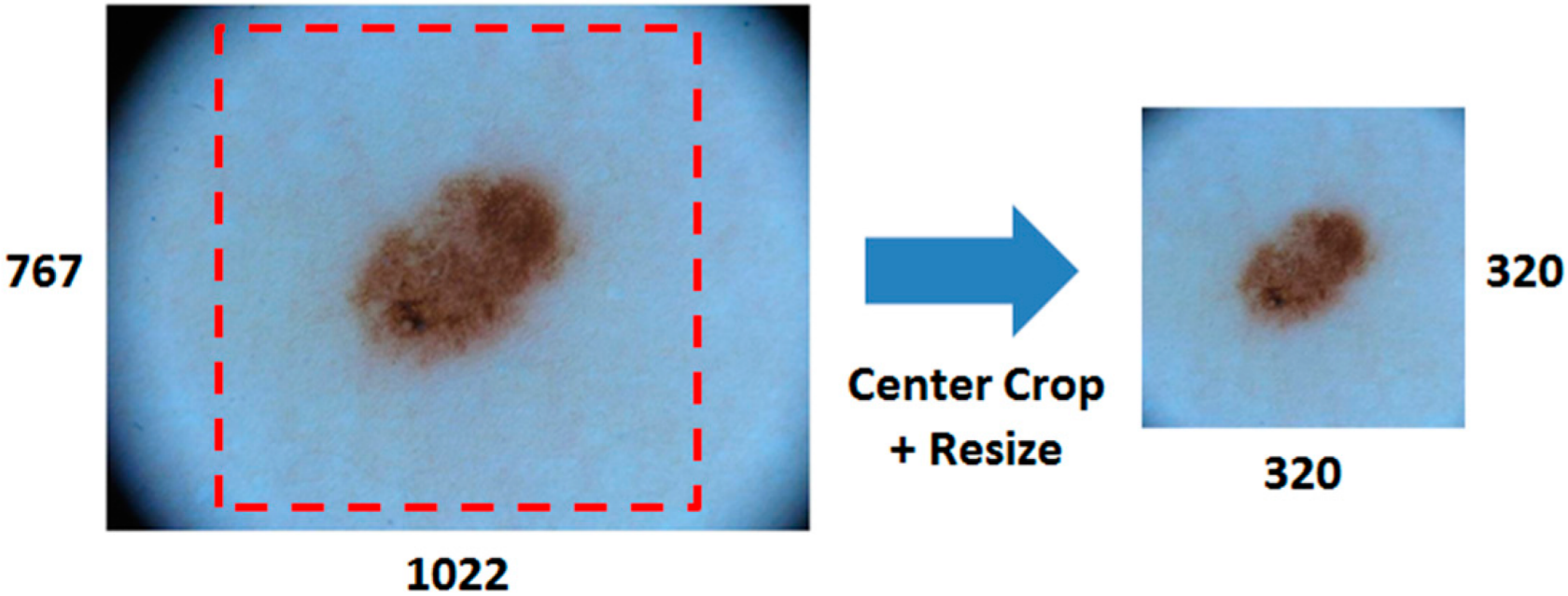
- Start with a ConvNet (e.g. Inception-V3) initialized with ImageNet pretrained weights;
- Remove the last fully-connected layers.
- Extract the 2048-D features by adding a global average pooling layer after the last convolutional layer.
- Adding the output layer for predicting vehicle models, which is a fully-connected layer.

```
    print('Loading InceptionV3 Weights ...')
    inception = InceptionV3(include_top=False, weights= None,
            input_tensor=None, input_shape=(IMG_WIDTH, IMG_HEIGHT, 3), pooling = 'avg')

    output = inception.get_layer(index = -1).output     # shape=(None, 1, 1, 2048)
    output = Dense(1024, name='features')(output)
    output = Dense(NBR_MODELS, activation='softmax', name='predictions')(output)
    model = Model(outputs = output, inputs = inception.input)

print('Training model begins...')
```

How to Resize Image to ConvNet's Input Size?

Question: Suppose the network's input shape is $299 \times 299 \times 3$, how to resize your training data into that shape?

- Simply resize arbitrary image into $299 \times 299 \times 3$, which will lead to distortion.
- A better way is to firstly scale the shorter side into 299 and then scale the longer side proportionally. Secondly, we crop the center region of the scaled image (we call this method *center-crop*).



Center-Crop: Code Snippet

```
def center_crop(x, center_crop_size):
    centerw, centerh = x.shape[0] // 2, x.shape[1] // 2
    halfw, halfh = center_crop_size[0] // 2, center_crop_size[1] // 2
    cropped = x[centerw - halfw : centerw + halfw,
                centerh - halfh : centerh + halfh, :]

    return cropped

def scale_byRatio(img_path, ratio=1.0, return_width=299, crop_method=center_crop):
    # Given an image path, return a scaled array
    img = cv2.imread(img_path)
    h = img.shape[0]
    w = img.shape[1]
    shorter = min(w, h)
    longer = max(w, h)
    img_cropped = crop_method(img, (shorter, shorter))
    img_resized = cv2.resize(img_cropped, (return_width, return_width),
                             interpolation=cv2.INTER_CUBIC)
    img_rgb = img_resized
    img_rgb[:, :, [0, 1, 2]] = img_resized[:, :, [2, 1, 0]]

    return img_rgb
```

Data Augmentation: Code Snippet

```
seq = iaa.Sequential(
    [
        # apply the following augmenters to most images
        iaa.Fliplr(0.5), # horizontally flip 50% of all images
        # crop images by -5% to 10% of their height/width
        sometimes(iaa.CropAndPad(
            percent=(-0.05, 0.1),
            pad_mode=ia.ALL,
            pad_cval=(0, 255)
        )),
        sometimes(iaa.Affine(
            scale={"x": (0.85, 1.15), "y": (0.85, 1.5)}, # scale images to 80-120% of their size, individually per axis
            translate_percent={"x": (-0.15, 0.15), "y": (-0.15, 0.15)}, # translate by -20 to +20 percent (per axis)
            rotate=(-15, 15), # rotate by -45 to +45 degrees
            shear=(-5, 5), # shear by -16 to +16 degrees
            order=[0, 1], # use nearest neighbour or bilinear interpolation (fast)
            cval=(0, 255), # if mode is constant, use a cval between 0 and 255
            mode=ia.ALL # use any of scikit-image's warping modes (see 2nd image from the top for examples)
        )),
        # execute 0 to 5 of the following (less important) augmenters per image
        # don't execute all of them, as that would often be way too strong
        iaa.SomeOf((0, 3),
            [
                iaa.OneOf([
                    iaa.GaussianBlur((0, 2.0)), # blur images with a sigma between 0 and 3.0
                    iaa.AverageBlur(k=(1, 5)), # blur image using local means with kernel sizes between 2 and 7
                    iaa.MedianBlur(k=(1, 5)), # blur image using local medians with kernel sizes between 2 and 7
                ]),

                iaa.AdditiveGaussianNoise(loc=0, scale=(0.0, 0.03*255), per_channel=0.5), # add gaussian noise to image
                iaa.OneOf([
                    iaa.Dropout((0.01, 0.1), per_channel=0.5), # randomly remove up to 10% of the pixels
                    iaa.CoarseDropout((0.03, 0.15), size_percent=(0.01, 0.03), per_channel=0.2),
                ]),
                iaa.Add((-10, 10), per_channel=0.5), # change brightness of images (by -10 to 10 of original value)

                iaa.ContrastNormalization((0.3, 1.0), per_channel=0.5), # improve or worsen the contrast
                iaa.Grayscale(alpha=(0.0, 1.0)),
                sometimes(iaa.ElasticTransformation(alpha=(0.5, 3.5), sigma=0.25)), # move pixels locally around (with
                sometimes(iaa.PiecewiseAffine(scale=(0.01, 0.05))), # sometimes move parts of the image around
            ],
```

keras_fit

fit

```
fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_split=0.0, validation
```

Trains the model for a fixed number of epochs (iterations on a dataset).

Arguments

- **x**: Numpy array of training data (if the model has a single input), or list of Numpy arrays (if the model has multiple inputs). If input layers in the model are named, you can also pass a dictionary mapping input names to Numpy arrays. `x` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **y**: Numpy array of target (label) data (if the model has a single output), or list of Numpy arrays (if the model has multiple outputs). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays. `y` can be `None` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **batch_size**: Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32.
- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided. Note that in conjunction with `initial_epoch`, `epochs` is to be understood as "final epoch". The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.
- **verbose**: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **validation_split**: Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the `x` and `y` data provided, before shuffling.
- **validation_data**: tuple `(x_val, y_val)` or tuple `(x_val, y_val, val_sample_weights)` on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. `validation_data` will override `validation_split`.
- **shuffle**: Boolean (whether to shuffle the training data before each epoch) or str (for 'batch'). 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when `steps_per_epoch` is not `None`.

keras fit_generator

fit_generator

```
fit_generator(self, generator, steps_per_epoch=None, epochs=1, verbose=1, callbacks=None, validation_data=None,
```

Trains the model on data generated batch-by-batch by a Python generator or an instance of `Sequence`.

The generator is run in parallel to the model, for efficiency. For instance, this allows you to do real-time data augmentation on images on CPU in parallel to training your model on GPU.

The use of `keras.utils.Sequence` guarantees the ordering and guarantees the single use of every input per epoch when using `use_multiprocessing=True`.

Arguments

- **generator**: A generator or an instance of `Sequence` (`keras.utils.Sequence`) object in order to avoid duplicate data when using multiprocessing. The output of the generator must be either
- a tuple `(inputs, targets)`
- a tuple `(inputs, targets, sample_weights)`. This tuple (a single output of the generator) makes a single batch. Therefore, all arrays in this tuple must have the same length (equal to the size of this batch). Different batches may have different sizes. For example, the last batch of the epoch is commonly smaller than the others, if the size of the dataset is not divisible by the batch size. The generator is expected to loop over its data indefinitely. An epoch finishes when `steps_per_epoch` batches have been seen by the model.
- **steps_per_epoch**: Integer. Total number of steps (batches of samples) to yield from `generator` before declaring one epoch finished and starting the next epoch. It should typically be equal to the number of samples of your dataset divided by the batch size. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire data provided, as defined by `steps_per_epoch`. Note that in conjunction with `initial_epoch`, `epochs` is to be understood as "final epoch". The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.
- **verbose**: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.

Example of keras fit_generator

Example

```
def generate_arrays_from_file(path):
    while True:
        with open(path) as f:
            for line in f:
                # create numpy arrays of input data
                # and labels, from each line in the file
                x1, x2, y = process_line(line)
                yield ({'input_1': x1, 'input_2': x2}, {'output': y})

model.fit_generator(generate_arrays_from_file('/my_file.txt'),
                    steps_per_epoch=10000, epochs=10)
```

Example of A Real generator

```
batch_index = 0
while True:
    current_index = (batch_index * batch_size) % N
    if N >= (current_index + batch_size):
        current_batch_size = batch_size
        batch_index += 1
    else:
        current_batch_size = N - current_index
        batch_index = 0
        if shuffle:
            random.shuffle(data_list)

    X_batch = np.zeros((current_batch_size, img_width, img_height, 3))
    Y_batch = np.zeros((current_batch_size, nbr_classes))

    for i in range(current_index, current_index + current_batch_size):
        line = data_list[i].strip().split(' ')
        #print line
        if return_label:
            label = int(line[-1])
        img_path = line[0]

        if random_scale:
            scale_ratio = random.uniform(0.9, 1.1)
        img = scale_byRatio(img_path, ratio=scale_ratio, return_width=img_width,
                                crop_method=crop_method)

        X_batch[i - current_index] = img
        if return_label:
            Y_batch[i - current_index, label] = 1

    if augment:
        X_batch = X_batch.astype(np.uint8)
        X_batch = seq.augment_images(X_batch)
```

Example of Using fit_generator

```
model.fit_generator(generator(train_data_lines, NBR_MODELS = NBR_MODELS,
                    batch_size = BATCH_SIZE, img_width = IMG_WIDTH,
                    img_height = IMG_HEIGHT, random_scale = RANDOM_SCALE,
                    shuffle = True, augment = True),
                    steps_per_epoch = steps_per_epoch, epochs = NBR_EPOCHS, verbose = 1,
                    validation_data = generator_batch(val_data_lines,
                    NBR_MODELS = NBR_MODELS, batch_size = BATCH_SIZE,
                    img_width = IMG_WIDTH, img_height = IMG_HEIGHT,
                    shuffle = False, augment = False),
                    validation_steps = validation_steps,
                    class_weight = class_weights, callbacks = [best_model, reduce_lr],
                    max_queue_size = 80, workers = 8, use_multiprocessing=True)
```

generator

Checkpoints, Learning Rate Strategy, Early Stopping

- We save model checkpoints periodically, e.g. before or after each epoch;
- We could set different values of learning rate in different training stages. A good rule of thumb is using higher learning rate in the first few epochs and decrease it afterwards;
- Early stopping is a practical technique to prevent from overfitting.

```
model_file_saved = "./models/attributes_models/InceptionV3_vehicleModelColor_facs=
# Define several callbacks

checkpoint = ModelCheckpoint(model_file_saved, verbose = 1)

reduce_lr = ReduceLROnPlateau(monitor='val_'+monitor_index, factor=0.5,
                              patience=3, verbose=1, min_lr=0.00001)

early_stop = EarlyStopping(monitor='val_'+monitor_index, patience=15, verbose=1)
```

Method 2: Multitask Leanring for Vehicle Models and Colors

Similar to method 1, except that the multitask leanring network has two branchs output. Please try to draw the whole framework by yourself.

Method 2: Code Snippet

- Start with a ConvNet (e.g. Inception-V3) initialized with ImageNet pretrained weights;
- Remove the last fully-connected layers;
- Extract the 2048-D features by adding a global average pooling layer after the last convolutional layer;
- Adding two output layers for predicting vehicle models and colors.

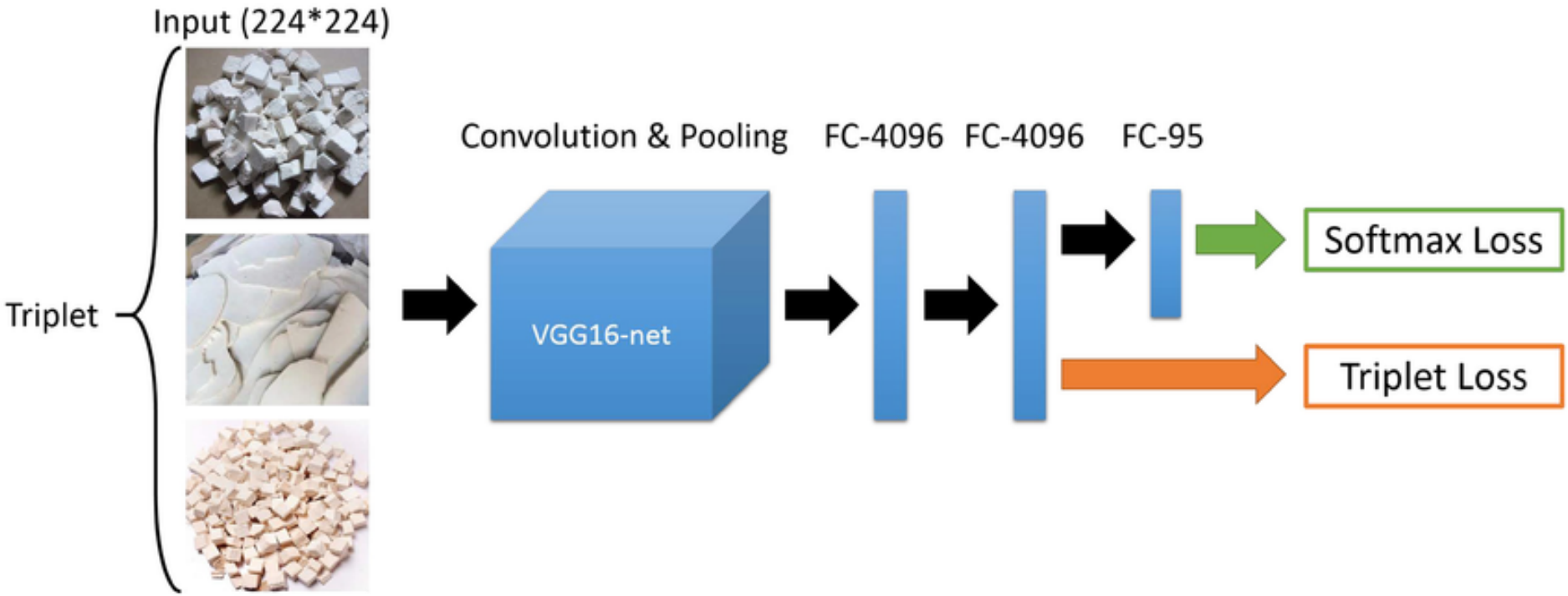
```
inception = InceptionV3(include_top=False, weights= None,
        input_tensor=None, input_shape=(IMG_WIDTH, IMG_HEIGHT, 3), pooling = 'avg')

f_base = inception.get_layer(index = -1).output     # shape=(None, 1, 1, 2048)

f_acs = Dense(1024, name='f_acs')(f_base)

f_model = Dense(NBR_MODELS, activation='softmax', name='predictions_model')(f_acs)

f_color = Dense(NBR_COLORS, activation='softmax', name='predictions_color')(f_acs)

model = Model(outputs = [f_model, f_color], inputs = inception.input)
```

Method 2: Code Snippet

- Set cross-entropy loss for the two output layers;
- Set different loss weights for two branches if necessary. For example, if we want to the ConvNet focus more on the model than color, we could set 0.6 to the model branch and 0.4 to the color one.

```
model.compile(loss=["categorical_crossentropy", "categorical_crossentropy"],
              #loss_weights = [0.6, 0.4],
              optimizer=optimizer, metrics=["accuracy"])
```

Method 3: Multi-task Learning = Triplet Loss + Cross-Entropy



Method 3: Implementation of Triplet Loss

```
def triplet_loss(vects):
    # f_anchor.shape = (batch_size, 256)
    f_anchor, f_positive, f_negative = vects
    # L2 normalize anchor, positive and negative, otherwise,
    # the loss will result in ''nan''!
    f_anchor = K.l2_normalize(f_anchors, axis = -1)
    f_positive = K.l2_normalize(f_positive, axis = -1)
    f_negative = K.l2_normalize(f_negative, axis = -1)

    dis_anchor_positive = K.sum(K.square(K.abs(f_anchor - f_positive)),
                                axis = -1, keepdims = True)

    dis_anchor_negative = K.sum(K.square(K.abs(f_anchor - f_negative)),
                                axis = -1, keepdims = True)
    loss = dis_anchor_positive + MARGIN - dis_anchor_negative
    return loss
```

Motivation of Repression Network (RepNet)

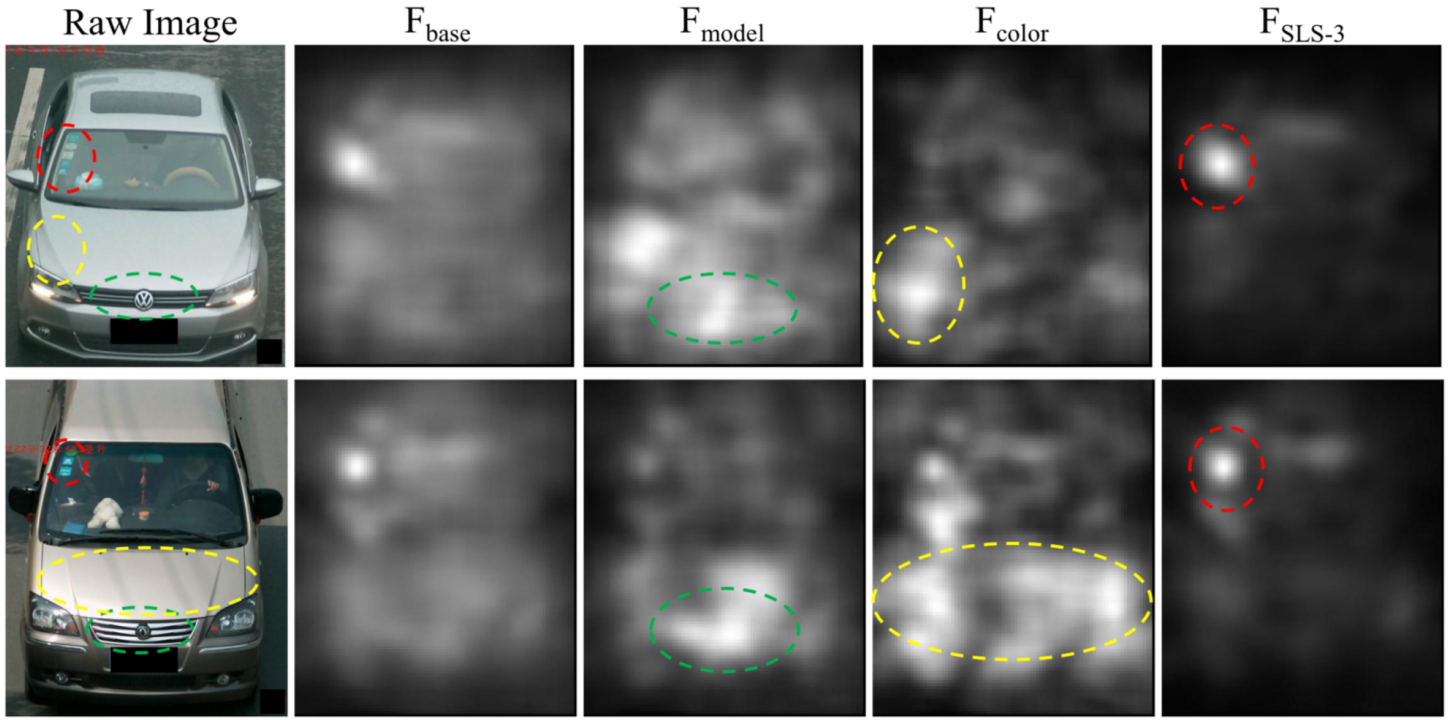


Figure 1. Saliency map of features from different layers in RepNet with PRL. The brighter an area is in the saliency map, the more information of the original image in that area is embedded in the given feature vector. The first column is the query image and column two to five shows the saliency maps of four feature vectors – F_{base}, F_{model}, F_{color} and F_{SLS-3}, respectively.

RepNet Architecture

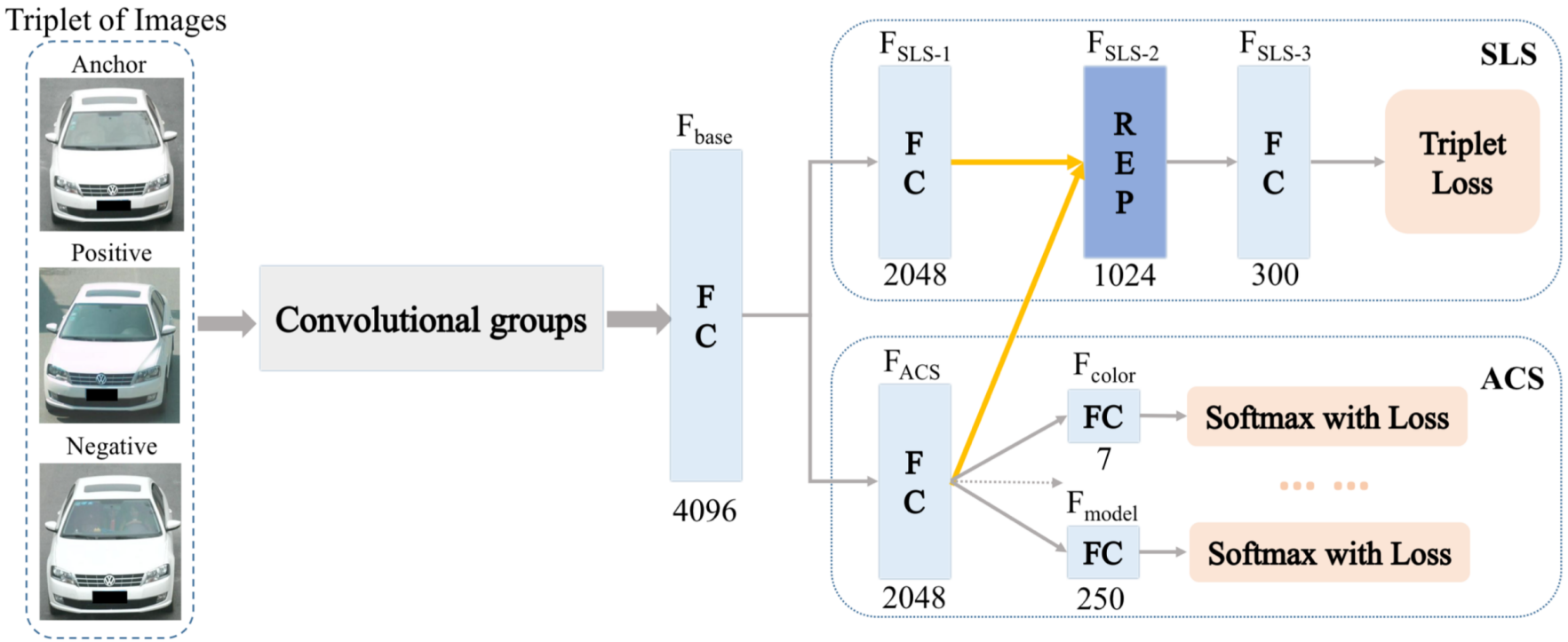


Figure 2. RepNet takes image triplets as input and feeds them through three networks, including convolutional groups, FC layers and Repression layers (REP), with shared weights. Names and dimensions of output features are listed above and under each layer. Repression layer takes two feature vectors F_{SLS-1} and F_{ACS} as input. Only the anchor image in each triplet is used for attribute classification, i.e. only the network for it has FC layers and loss functions after F_{ACS}.

Three Types of Repression Layer

- Product Repression Layer (PRL).

$$F_{SLS-2} = W_{PRL}^{T}(F_{SLS-1} \circ F_{ACS})$$

- Subtractive Repression Layer (SRL).

$$F_{SLS-2} = W_{SRL}^{T}(F_{SLS-1} - F_{ACS})$$

- Concatenated Repression Layer (CRL) - best performance.

$$F_{SLS-2} = W_{CRL}^{T}[F_{SLS-1}; F_{ACS}]$$

Visualization of Repression Layer

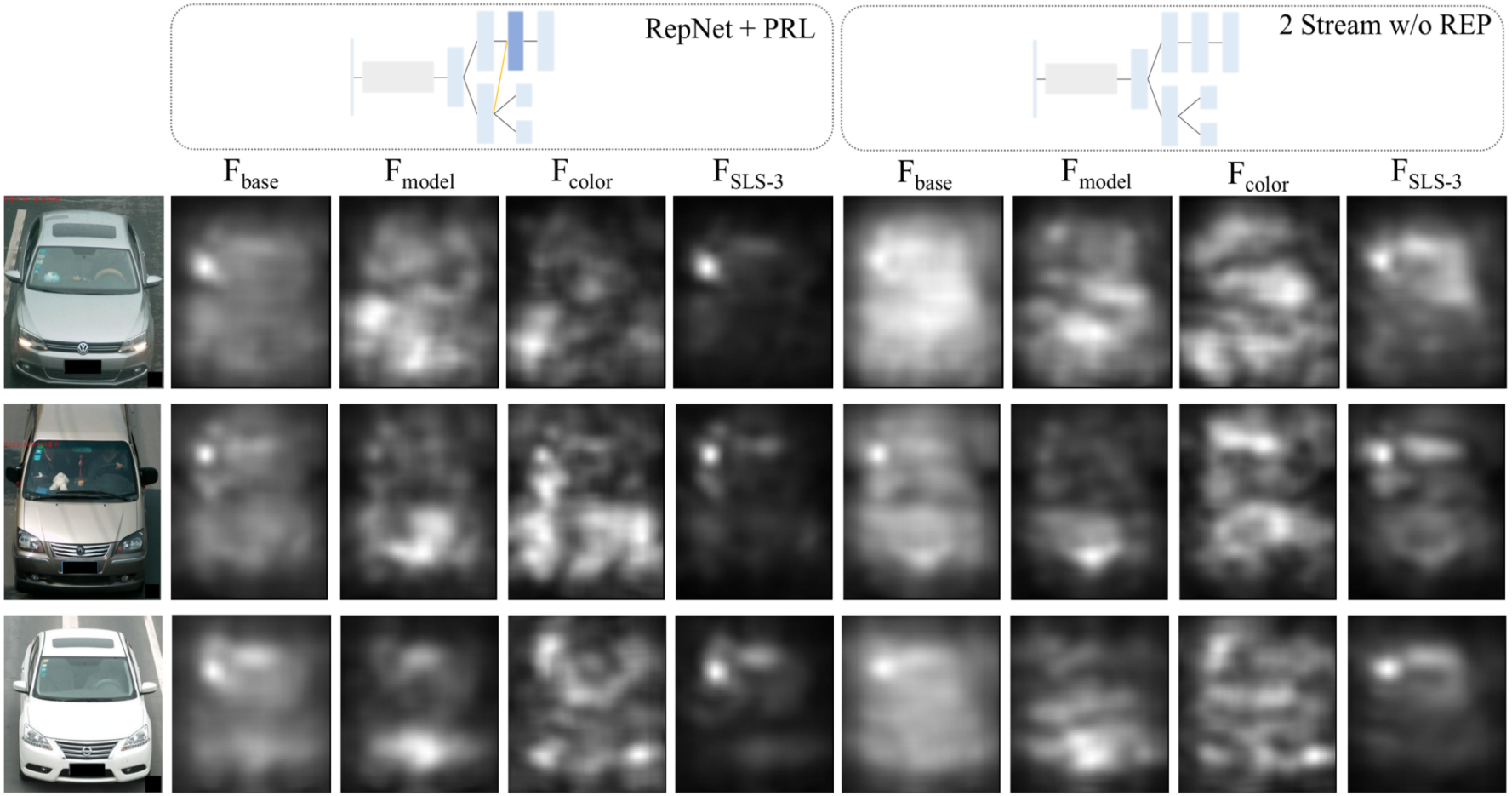


Figure 3. Saliency map of features from different layers in RepNet with PRL and a network with same architecture but repression layer. The first column is the query image and the following two groups of four columns shows the saliency maps of four feature vectors – F_{base}, F_{model}, F_{color} and F_{SLS-3}, respectively learned from two networks.

Method 3: Implementation of RepNet

```
# Begin with Attributes pretrained weights.
attributes_branch = load_model('./models/attributes_models/InceptionV3_vehicleModelColor_fa
#attributes_branch.summary()
attributes_branch.get_layer(name = 'global_average_pooling2d_1').name = 'f_base'
f_base = attributes_branch.get_layer(name = 'f_base').output  # 1024-D

anchor = attributes_branch.input
positive = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 3), name='positive')
negative = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 3), name='negative')

# Attributes Branch
f_acs = attributes_branch.get_layer(name = 'f_acs').output
f_model = attributes_branch.get_layer(name = 'predictions_model').output
f_color = attributes_branch.get_layer(name = 'predictions_color').output

# Similarity Learning Branch
f_sls1 = Dense(1024, name = 'sls1')(f_base)
f_sls2 = concatenate([f_sls1, f_acs], axis = -1, name = 'sls1_concatenate')  # 1024-D
# The author said that only layer ``SLS_2'' is applied ReLU since non-linearity
# would disrupt the embedding learned in the layer ``SLS_1''.
#f_sls2 = Activation('relu', name = 'sls1_concatenate_relu')(f_sls2)
f_sls2 = Dense(1024, name = 'sls2')(f_sls2)
f_sls2 = Activation('relu', name = 'sls2_relu')(f_sls2)
# Non-linearity ?
f_sls3 = Dense(256, name = 'sls3')(f_sls2)
sls_branch = Model(inputs = attributes_branch.input, outputs = f_sls3)
f_sls3_anchor = sls_branch(anchor)
f_sls3_positive = sls_branch(positive)
f_sls3_negative = sls_branch(negative)

loss = Lambda(triplet_loss,
              output_shape=(1, ))([f_sls3_anchor, f_sls3_positive, f_sls3_negative])

model = Model(inputs = [anchor, positive, negative], outputs = [f_model, f_color, loss])
```

Sampling Positives and Negatives

- The strategy of sampling positives and negatives is critical to the triplet network;
- Anchors and positives share with the same vehicle ID;
- We could randomly pick a vehicle photo from the database as the negative;
- The random sampling approch does not contribute to the network learning;
- A better approach is to discover those "hard negatives";
- Hard negatives are usually vehicle photos with the same model and color, but with different vehicle IDs;
- It is better to sampling different psotives and negatives during training progress (in different iterations).

Future Work Tip: Inspection Marks

- Inspection marks are unique features for vehicles.
- Train an object detection model (e.g. Faster R-CNN) to detect inspection marks locations.
- Train another deep netowrk which predicts the similarity of a pair of inspection marks. A feasible model is the siamese network.
- The inspection marks similarities can be considered as a local similarity which further improves ranking performance.