



ETH Zürich
Eidgenössische Technische Hochschule Zürich

Computer Vision
Department of Computer Science

Lab 4:
Object Recognition

Christos Dimopoulos - 23-941-834
cdimopoulos@student.ethz.ch

November, 2023

Contents

Task 1: Bag-of-Words Classifier	2
1. Local Feature Extraction	2
2. Codebook construction	4
3. Bag-of-words Vector Encoding	5
4. Nearest Neighbor Classification	6
Task 2: CNN-based Classifier	7

Task 1: Bag-of-Words Classifier

In the first part of the lab exercise, we implement a Bag of Visual Words Classifier that determines whether an input image contains a car (label 1) or not (label 0). The entire procedure can be summarised on the following steps:

- **Local Feature Extraction** which involves a simple local feature point detector (points on a grid) and a HOG feature descriptor.
- **Codebook construction** via the k-means clustering algorithm
- **Bag-of-words Vector Encoding** as a representation that records which visual words are present in each image.
- **Nearest Neighbor Classification** during inference time to classify the test images.

Task 1: Bag of Visual Words

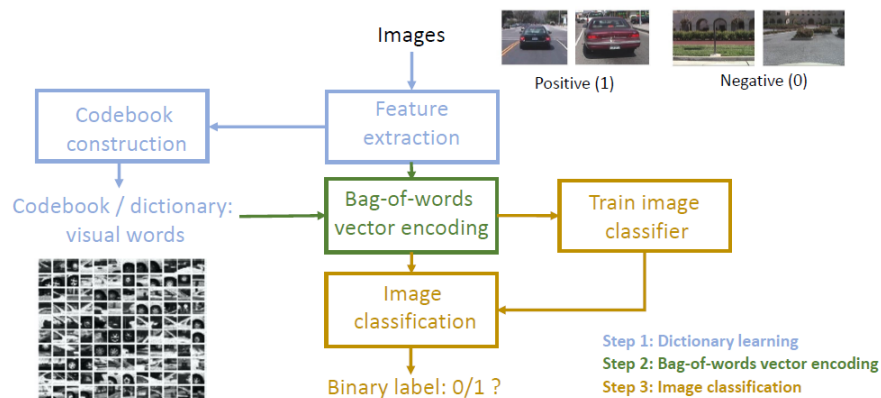


Figure 1: BoW Classifier Procedure.

We analyze each step in our code implementation below. The initial parameter values for k-means are $k=10$ clusters and $numiter=200$.

1. Local Feature Extraction

First of all, we implement a very simple local feature point detector: points on a grid. Within the function grid points, we compute a regular grid that fits the given input image, leaving a border of 8 pixels in each image dimension. The parameters $nPointsX = 10$ and $nPointsY = 10$ define the granularity of the grid in the x and y dimensions. The function shall return $nPointsX \cdot nPointsY$ 2D grid points.

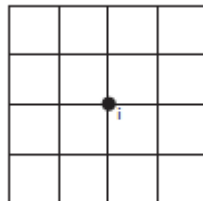


Figure 2: Grid point i and surrounding cells.

```

def grid_points(img, nPointsX, nPointsY, border):
    """
    :param img: input gray img, numpy array, [h, w]
    :param nPointsX: number of grids in x dimension
    :param nPointsY: number of grids in y dimension
    :param border: leave border pixels in each image dimension
    :return: vPoints: 2D grid point coordinates, numpy array, [nPointsX*nPointsY, 2]
    """

    # todo
    h, w = img.shape
    step_x = (w - 2 * border) / (nPointsX - 1)
    step_y = (h - 2 * border) / (nPointsY - 1)

    vPoints = np.zeros((nPointsX * nPointsY, 2), dtype=np.int) # numpy array,
        [nPointsX*nPointsY, 2]

    for i in range(nPointsX):
        for j in range(nPointsY):
            x = int(border + i * step_x)
            y = int(border + j * step_y)
            vPoints[i * nPointsY + j] = [x, y]

    return vPoints

```

Next, we describe each feature (grid point) by a local descriptor. We use the well known **histogram of oriented gradients (HOG) descriptor**. The following function computes one 128-dimensional descriptor for each of the N 2-dimensional grid points (contained in the $N \times 2$ input argument `vPoints`). The descriptor is defined over a 4×4 set of cells placed around the grid point i .

For each cell (containing $\text{cellWidth} \times \text{cellHeight}$ pixels, which we choose to be $\text{cellWidth} = \text{cellHeight} = 4$), an 8-bin histogram over the orientation of the gradient at each pixel within the cell is created. To compute the angles of each orientation we use the **arctan** function, while for the creation of histograms we use the function *histogram* of the *numpy* library. Then we concatenate the 16 resulting histograms (one for each cell) to produce a 128 dimensional vector for every grid point. Finally, the function should return a $N \times 128$ dimensional feature matrix.

```

def descriptors_hog(img, vPoints, cellWidth, cellHeight):
    nBins = 8
    w = cellWidth
    h = cellHeight

    grad_x = cv2.Sobel(img, cv2.CV_16S, dx=1, dy=0, ksize=1)
    grad_y = cv2.Sobel(img, cv2.CV_16S, dx=0, dy=1, ksize=1)

    descriptors = [] # list of descriptors for the current image, each entry is one
        128-d vector for a grid point
    for i in range(len(vPoints)):
        center_x = round(vPoints[i, 0])
        center_y = round(vPoints[i, 1])

        desc = []
        for cell_y in range(-2, 2):
            for cell_x in range(-2, 2):
                start_y = center_y + (cell_y) * h
                end_y = center_y + (cell_y + 1) * h

```

```

start_x = center_x + (cell_x) * w
end_x = center_x + (cell_x + 1) * w

# todo
# compute the angles
# compute the histogram
angles = np.arctan2(grad_y[start_y:end_y, start_x:end_x],
                    grad_x[start_y:end_y, start_x:end_x])
histogram, _ = np.histogram(angles, bins=nBins, range=(-np.pi, np.pi))

# Concatenate the histograms
desc.extend(histogram)

descriptors.append(desc)

descriptors = np.asarray(descriptors) # [nPointsX*nPointsY, 128], descriptor for
    the current image (100 grid points)
return descriptors

```

2. Codebook construction

In order to capture the appearance variability within an object category, we first need to construct a visual vocabulary (or appearance codebook). For this we cluster the large set of all local descriptors from the training images into a small number of visual words (which form the entries of the codebook), using the **K-means clustering algorithm**. The function *create_codebook* takes as input the name of a directory, loads each image in it in turn, computes grid points for it, extracts descriptors around each grid point, and clusters all the descriptors with K-Means. It then returns the found cluster centers.

```

def create_codebook(nameDirPos, nameDirNeg, k, numiter):
    """
    :param nameDirPos: dir to positive training images
    :param nameDirNeg: dir to negative training images
    :param k: number of kmeans cluster centers
    :param numiter: maximum iteration numbers for kmeans clustering
    :return: vCenters: center of kmeans clusters, numpy array, [k, 128]
    """
    vImgNames = sorted(glob.glob(os.path.join(nameDirPos, '*.png')))
    vImgNames = vImgNames + sorted(glob.glob(os.path.join(nameDirNeg, '*.png')))

    nImgs = len(vImgNames)

    cellWidth = 4
    cellHeight = 4
    nPointsX = 10
    nPointsY = 10
    border = 8

    vFeatures = [] # list for all features of all images (each feature: 128-d, 16
                    histograms containing 8 bins)
    # Extract features for all image
    for i in tqdm(range(nImgs)):
        # print('processing image {} ...'.format(i+1))
        img = cv2.imread(vImgNames[i]) # [172, 208, 3]
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # [h, w]

```

```

    # Collect local feature points for each image, and compute a descriptor for
    # each local feature point
    # todo
    vPoints = grid_points(img, nPointsX, nPointsY, border)
    descriptors = descriptors_hog(img, vPoints, cellWidth, cellHeight)

    # Append the descriptors to the list
    vFeatures.append(descriptors)

vFeatures = np.asarray(vFeatures) # [n_imgs, n_vPoints, 128]
vFeatures = vFeatures.reshape(-1, vFeatures.shape[-1]) # [n_imgs*n_vPoints, 128]
print('number of extracted features: ', len(vFeatures))

# Cluster the features using K-Means
print('clustering ...')
kmeans_res = KMeans(n_clusters=k, max_iter=numiter).fit(vFeatures)
vCenters = kmeans_res.cluster_centers_ # [k, 128]
return vCenters

```

3. Bag-of-words Vector Encoding

Using the appearance codebook created at the previous step we will represent each image as a histogram of visual words. Firstly, we fill the function *bow_histogram* that computes a bag-of-words histogram corresponding to a given image. The function should take as input a set of descriptors extracted from one image and the codebook of cluster centers. To compute the histogram to be returned, we assign the descriptors to the cluster centers and count how many descriptors are assigned to each cluster.

```

def bow_histogram(vFeatures, vCenters):
    """
    :param vFeatures: MxD matrix containing M feature vectors of dim. D
    :param vCenters: NxD matrix containing N cluster centers of dim. D
    :return: histo: N-dim. numpy vector containing the resulting BoW activation
            histogram.
    """
    histo = np.zeros(vCenters.shape[0])

    # Assign each descriptor to the closest cluster center
    labels = np.argmin(np.linalg.norm(vFeatures[:, None] - vCenters, axis=2), axis=1)

    # Count how many descriptors are assigned to each cluster
    for label in labels:
        histo[label] += 1

    return histo

```

Next up, we read in all training examples (images) from a given directory and computes a bag-of-words histogram for each. The output should be a matrix having the number of rows equal to the number of training examples and number of columns equal to the size of the codebook (number of cluster centers). This way, we process all positive training examples from the directory **cars-training-pos** and all negative training examples from the directory **cars-training-neg**. We collect the resulting histograms in the rows of the matrices *vBoWPos* and *vBoWNeg*.

```

def create_bow_histograms(nameDir, vCenters):
    """
    :param nameDir: dir of input images
    :param vCenters: kmeans cluster centers, [k, 128] (k is the number of cluster
        centers)
    :return: vBoW: matrix, [n_imgs, k]
    """
    vImgNames = sorted(glob.glob(os.path.join(nameDir, '*.png')))
    n_imgs = len(vImgNames)

    cellWidth = 4
    cellHeight = 4
    nPointsX = 10
    nPointsY = 10
    border = 8

    # Extract features for all images in the given directory
    vBoW = []
    for i in tqdm(range(n_imgs)):
        # print('processing image {} ...'.format(i + 1))
        img = cv2.imread(vImgNames[i]) # [172, 208, 3]
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # [h, w]

        # todo
        # Collect local feature points for each image, and compute a descriptor for
        # each local feature point
        vPoints = grid_points(img, nPointsX, nPointsY, border)
        descriptors = descriptors_hog(img, vPoints, cellWidth, cellHeight)

        # Compute bag-of-words histogram for the image
        bow_hist = bow_histogram(descriptors, vCenters)
        vBoW.append(bow_hist)

    vBoW = np.asarray(vBoW) # [n_imgs, k]
    return vBoW

```

4. Nearest Neighbor Classification

For inference, we build a bag-of-words image classifier using the **nearest neighbor principle**. For classifying a new test image, we compute its bag-of-words histogram, and assign it to the category of its nearest neighbor training histogram. More specifically, we compute the bag-of-words histogram of a test image to the positive and negative training examples and return the label 1 (car) or 0 (no car), based on the label of the nearest-neighbour.

```

def bow_recognition_nearest(histogram, vBoWPos, vBoWNeg):
    """
    :param histogram: bag-of-words histogram of a test image, [1, k]
    :param vBoWPos: bag-of-words histograms of positive training images, [n_imgs, k]
    :param vBoWNeg: bag-of-words histograms of negative training images, [n_imgs, k]
    :return: sLabel: predicted result of the test image, 0(without car)/1(with car)
    """

    # Compute the Euclidean distances between the test image histogram and the
    # histograms of positive and negative training images
    DistPos = np.linalg.norm(vBoWPos - histogram, axis=1)

```

```

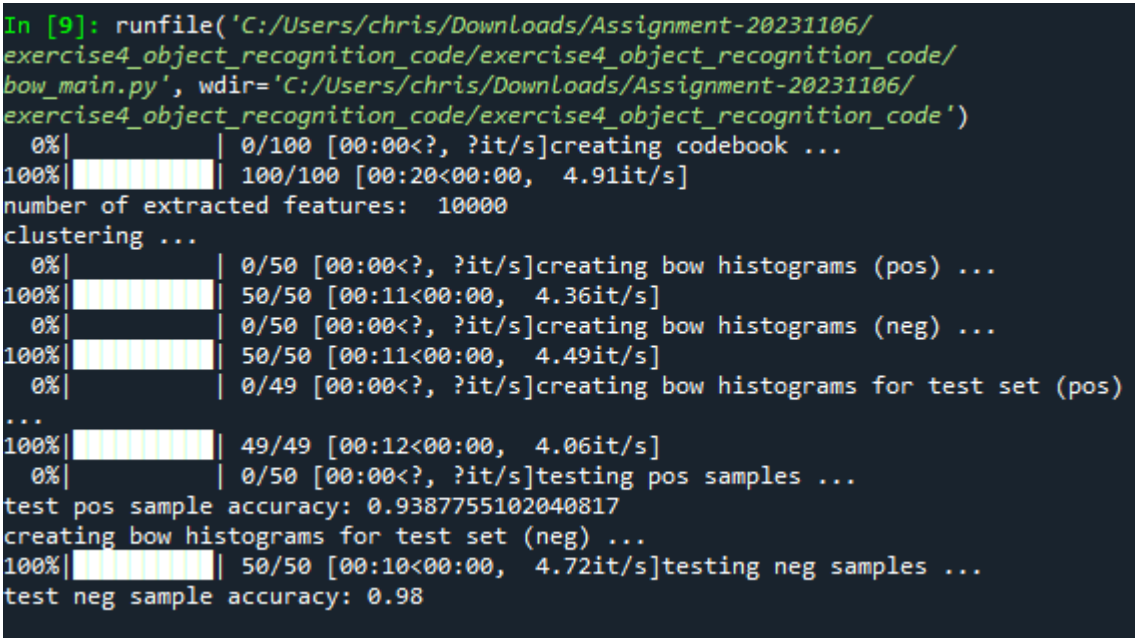
DistNeg = np.linalg.norm(vBoWNeg - histogram, axis=1)

# Find the nearest neighbor in the positive and negative sets and decide based
# on this neighbor
minDistPos = np.min(DistPos)
minDistNeg = np.min(DistNeg)

if (minDistPos < minDistNeg):
    sLabel = 1
else:
    sLabel = 0
return sLabel

```

Having completed the whole procedure, we run the BoW Classification algorithm on the data given for cars. The parameters for the k-means clustering algorithm are defined as $k=10$ clusters and $numiter=200$. After running the algorithm, we achieve **test accuracy 0.9387 for positive samples and 0.98 for negative samples**, which is pretty satisfactory.



```

In [9]: runfile('C:/Users/chris/Downloads/Assignment-20231106/
exercise4_object_recognition_code/exercise4_object_recognition_code/
bow_main.py', wdir='C:/Users/chris/Downloads/Assignment-20231106/
exercise4_object_recognition_code/exercise4_object_recognition_code')
0%|          | 0/100 [00:00<?, ?it/s]creating codebook ...
100%|██████████| 100/100 [00:20<00:00, 4.91it/s]
number of extracted features: 10000
clustering ...
0%|          | 0/50 [00:00<?, ?it/s]creating bow histograms (pos) ...
100%|██████████| 50/50 [00:11<00:00, 4.36it/s]
0%|          | 0/50 [00:00<?, ?it/s]creating bow histograms (neg) ...
100%|██████████| 50/50 [00:11<00:00, 4.49it/s]
0%|          | 0/49 [00:00<?, ?it/s]creating bow histograms for test set (pos)
...
100%|██████████| 49/49 [00:12<00:00, 4.06it/s]
0%|          | 0/50 [00:00<?, ?it/s]testing pos samples ...
test pos sample accuracy: 0.9387755102040817
creating bow histograms for test set (neg) ...
100%|██████████| 50/50 [00:10<00:00, 4.72it/s]testing neg samples ...
test neg sample accuracy: 0.98

```

Figure 3: BoW Classification Results.

Task 2: CNN-based Classifier

For the second task, we implement a simplified version of the VGG image classification network on CIFAR-10 dataset, which includes 10 image classes, with 50000 training images, from which we split 5000 images for validation, and 10000 testing images, of resolution 32×32 . The simplified VGG network architecture is shown in the following table. For the training of VGG we opt for batch size 128, 40 epochs, learning rate 0.0001, 512 as feature number for the first linear layer and Adam Optimizer and Cross-Entropy Loss.

Block Name	Layers	Output Size
conv_block1	ConvReLU (k=3) + MaxPool2d(k=2)	[bs, 64, 16, 16]
conv_block2	ConvReLU (k=3) + MaxPool2d(k=2)	[bs, 128, 8, 8]
conv_block3	ConvReLU (k=3) + MaxPool2d(k=2)	[bs, 256, 4, 4]
conv_block4	ConvReLU (k=3) + MaxPool2d(k=2)	[bs, 512, 2, 2]
conv_block5	ConvReLU (k=3) + MaxPool2d(k=2)	[bs, 512, 1, 1]
classifier	Linear+ReLU+Dropout+Linear	[bs, 10]

Figure 4: Simplified VGG architecture.

After the completion of 40 training epochs, the loss reaches the value of **0.3578** while we get **training accuracy of 83.96%**. For inference, we achieve **test accuracy of 82.15%** on the CIFAR-10 Dataset, which is satisfying enough.

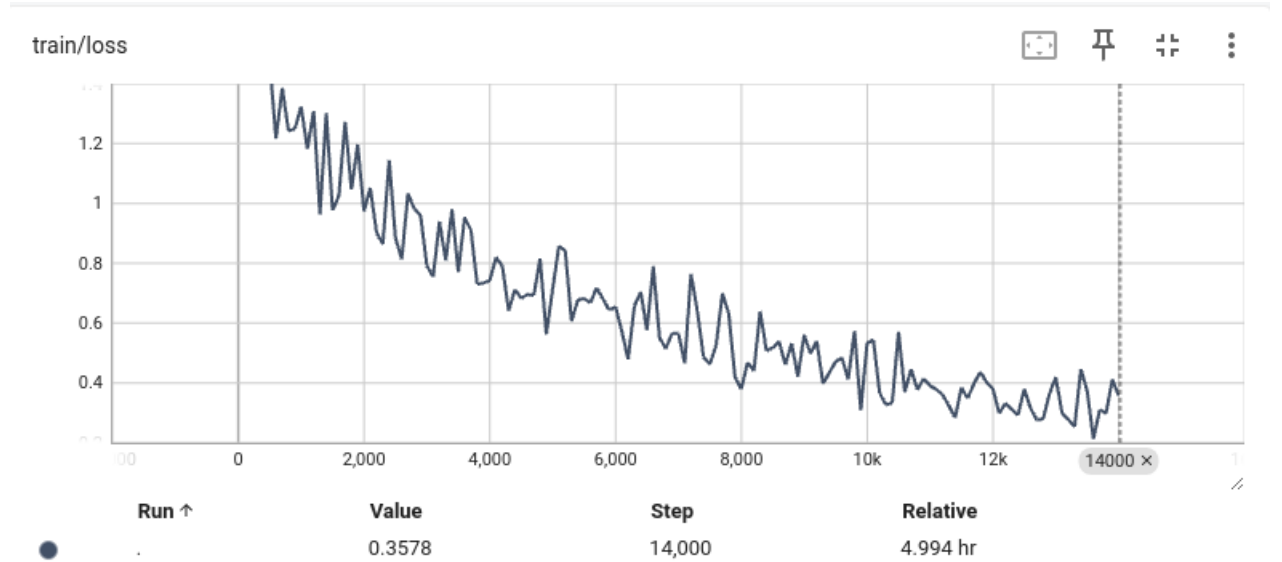


Figure 5: Training Loss for 40 epochs.

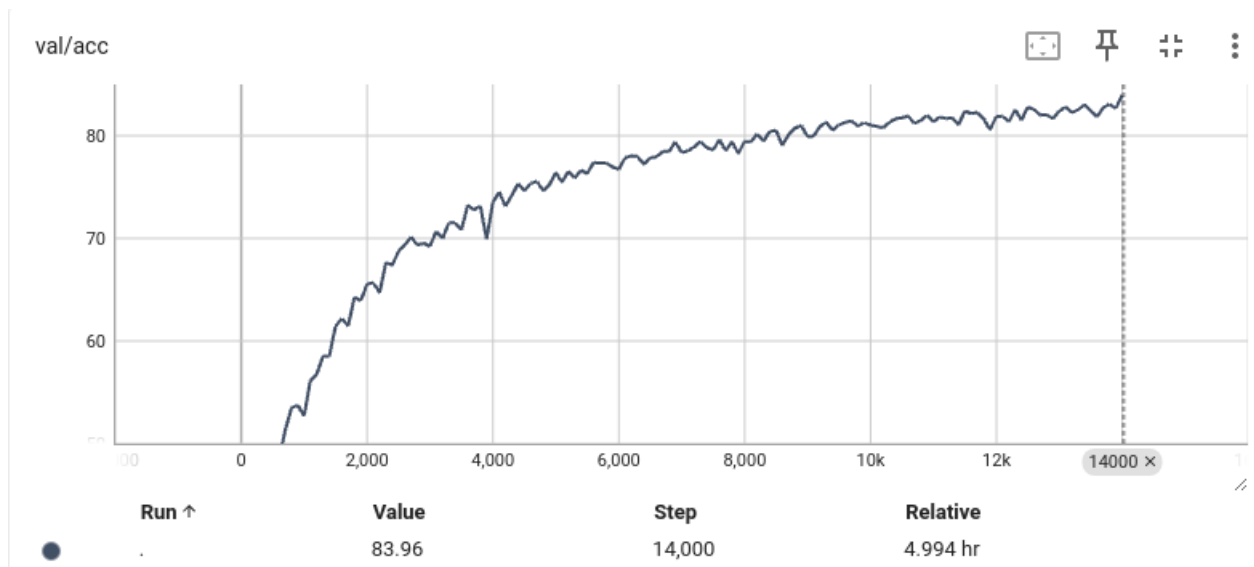


Figure 6: Training Accuracy for 40 epochs.