Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Vision
and Geometry Lab

ETH Zurich
Computer Science Department

Computer Vision

Lab Assignment - Object Recognition

Lara Nonino - lnonino@student.ethz.ch

November 2022

Student ID:
22-937-833

Section 1

Bag-of-words Classifier

The *Bag-of-words Classifier* is an algorithm for category recognition. This approach computes the distribution (histogram) of the visual words found in the query image and compares this distribution to those found in the training images.
In this section, we were asked to implement a bag-of-words image classifier that decides whether a test image contains a car (back view) or not.

1.1 Local Feature Extraction

1.1.1 Feature detection - feature points on a grid

To begin with, it was necessary to implement a local feature detector.
Within the function `grid_points(img, nPointsX, nPointsY, border)` we were asked to compute a regular grid that fits the given input image (`img`), leaving a border of `border` pixels in each image dimension. The parameters `nPointX` and `nPointY` define the granularity of the grid in the x and y dimensions. Following these instructions, the horizontal and vertical spacing between each grid point were found to be

$$stepX = \frac{w - 2 * border - 1}{nPointsX - 1} \qquad stepY = \frac{h - 2 * border - 1}{nPointsY - 1}$$

where $[h, w]$ are the dimentions of the input image. The implemented function is reported here

```
def grid_points(img, nPointsX, nPointsY, border):

    h = img.shape[0]
    w = img.shape[1]

    stepX = (w - 2 * border - 1) // (nPointsX - 1)
    stepY = (h - 2 * border - 1) // (nPointsY - 1)

    Px = np.arange(border, w - border, stepX)
    Py = np.arange(border, h - border, stepY)

    vPoints = np.transpose([np.tile(Px, len(Py)), np.repeat(Py, len(Px))])

    return vPoints
```

As you can notice, to select the coordinates of each grid point (which are the output of the function), I choose to select all the vertical and horizontal coordinates first (`Px` and `Py`) and than compute their Cartesian product (`np.transpose(...)`).

1.1.2 Feature description - histogram of oriented gradients

Next, we were asked to use the histogram of oriented gradients (HOG) descriptor to describe each feature (grid point).

The descriptor is defined over a 4×4 set of cells placed around the grid point i. The dimensions of the cells are given as parameter to the function `descriptors_hog`. The whole image and the grid points are also passed as inputs to the function.

More precisely, we were asked to create an 8-bin histogram over the orientation of the gradient at each pixel within a cell. Once this was done, I concatenated all the 16 histograms to each other (each point is surrounded by 4×4 cells). In this way, the function `descriptors_hog` computes one 128-dimensional descriptor for each grid point and returns a N $\times$ 128 dimensional feature matrix, where N is the number of grid points.

To compute an 8-bin histogram of oriented gradients, I firstly computed the orientation of a gradient using `np.arctan2(`$\nabla_x I, \nabla_y I$`)`. Since this returns an angle between $[-\pi, +\pi]$, to obtain the same angles between $[0, \pi]$ I computed the remainder of the division between the angle found and π.
Once this was done, I had to find the bin of each angle. This means that each angle between $[n \times \frac{\pi}{4}, (n+1) \times \frac{\pi}{4}]$ is approximated to $n \times \frac{\pi}{4}$ and its index results to be n.

```
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 = []
    for i in range(len(vPoints)):
        center_x = round(vPoints[i, 0])
        center_y = round(vPoints[i, 1])

        desc = np.empty(0)
        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

                y_idx = np.arange(start_y, end_y)
                x_idx = np.arange(start_x, end_x)

                #phases between [-pi, +pi] converted to [0, 2pi]
                angles = np.arctan2(grad_y[y_idx, x_idx], grad_x[y_idx, x_idx]) % (2 * np.pi)

                #convert the phases to indeces 0, ..., 7
                angles_idx = (np.round(angles * 4 / np.pi)).astype(np.intc)
                angles_idx[np.where(angles_idx == 8)] = 0

                #histogram per each point of the cell
                hist = np.histogram(angles_idx, bins=8, range=(0, 7))[0]

                desc = np.append(desc, hist)

        descriptors.append(desc)

    descriptors = np.asarray(descriptors)
```

```
    return descriptors
```

1.2 Codebook construction

Now that we have the descriptors for each image of the training set, it is necessary to capture the appearance variability within an object category. To do so, we were asked to construct an *appearance codebook*. This consists in clustering the large set of all local descriptors (which are $N_{images} \times N_{gridpoints}$) into a small number of visual words (which form the entries of the codebook). In order to do this, we were suggested to advantage of the K-means clustering algorithm already implemented by `sklearn`.

```
def create_codebook(nameDirPos, nameDirNeg, k, numiter):
    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 = []
    # 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]

        vPoints = grid_points(img, nPointsX, nPointsY, border)
        vFeatures.append(descriptors_hog(img, vPoints, cellWidth, cellHeight))

    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
```

1.3 Bag-of-words Vector Encoding

Thanks to the *appearance codebook* created, we can associate each image with a *bag-of-words* representation, which means that each image is represented as a histogram of visual words. To reach this result I proceeded as follows.

1.3.1 Bag-of-Words histogram

Firstly, I implemented the function `bow_histogram` that computes the bag-of-words histogram corresponding to a given image. The function takes a set of descriptors extracted from a single image (`vFeatures`) and the codebook of cluster centers (`vCenters`) as inputs. To compute the histogram, each descriptor is assigned to a cluster. The result is the count of descriptor per each cluster.

```
def bow_histogram(vFeatures, vCenters):

    N = vCenters.shape[0]
    Idx = findnn(vFeatures, vCenters)[0]
    histo = np.histogram(Idx, bins=N)[0]

    return histo
```

Note that the number of bins of the histogram is equal to the number of clusters. Moreover, the provided function `findnn` was used to find the closest point among a set of given input.

1.3.2 Processing a directory with training examples

Next, `create_bow_histograms` reads in all training images and computes the histogram for each.

```
def create_bow_histograms(nameDir, vCenters):
    vImgNames = sorted(glob.glob(os.path.join(nameDir, '*.png')))
    nImgs = 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(nImgs)):
        # print('processing image {} ...'.format(i + 1))
        img = cv2.imread(vImgNames[i])  # [172, 208, 3]
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)  # [h, w]

        vPoints = grid_points(img, nPointsX, nPointsY, border)
        vFeatures = descriptors_hog(img, vPoints, cellWidth, cellHeight)
        hist = bow_histogram(vFeatures, vCenters)
        vBoW.append(hist)

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

As you can see, the output is a matrix having the number of rows equal to the number of training images and the number of columns equal to the number of clusters.

1.4 Nearest Neighbor Classification

Once that we have the bag of words histogram for each training image, it is necessary to calculate the bag of words histogram for unseen test images (training images are already labeled). Once this is done, we can find the image of the training set whose histogram is the "closest" to the histogram of the new image. The label of the test image will be the same of the selected training image.

```
def bow_recognition_nearest(histogram,vBoWPos,vBoWNeg):
    DistPos, DistNeg = None, None

    DistPos = np.min(findnn(histogram, vBoWPos)[1])
    DistNeg = np.min(findnn(histogram, vBoWNeg)[1])

    if (DistPos < DistNeg):
        sLabel = 1
```

```
    else:
        sLabel = 0
    return sLabel
```

Note that the training images are divided in positive and negative training examples. As you can see, if the distance between the histograms is smaller between an image where a car is present (positive training example), the label will be 1, otherwise 0.

1.5 Results

The results found are here reported. Note that increasing the number of clusters (k) and the number of iteration it is possible to obtain a better accuracy.

```
/Users/lara/opt/anaconda3/envs/cv-lab04/bin/python /Users/lara/Desktop/ETH/courses/Computer Vison/lab04/exercise4_object_recognition_code/bow_main.py
creating codebook ...
100%|██████████| 100/100 [00:11<00:00,  8.39it/s]
number of extracted features:  10000
clustering ...
creating bow histograms (pos) ...
100%|██████████| 50/50 [00:06<00:00,  8.09it/s]
  0%|          | 0/50 [00:00<?, ?it/s]creating bow histograms (neg) ...
100%|██████████| 50/50 [00:06<00:00,  8.10it/s]
  0%|          | 0/49 [00:00<?, ?it/s]creating bow histograms for test set (pos) ...
100%|██████████| 49/49 [00:06<00:00,  8.07it/s]
testing pos samples ...
test pos sample accuracy: 0.9591836734693877
creating bow histograms for test set (neg) ...
100%|██████████| 50/50 [00:06<00:00,  8.12it/s]
testing neg samples ...
test neg sample accuracy: 0.84
```

Figure 1.1: k=10, numiter=10

```
/Users/lara/opt/anaconda3/envs/cv-lab04/bin/python /Users/lara/Desktop/ETH/courses/Computer Vison/lab04/exercise4_object_recognition_code/bow_main.py
creating codebook ...
100%|██████████| 100/100 [00:12<00:00,  8.27it/s]
number of extracted features:  10000
clustering ...
creating bow histograms (pos) ...
100%|██████████| 50/50 [00:06<00:00,  7.95it/s]
  0%|          | 0/50 [00:00<?, ?it/s]creating bow histograms (neg) ...
100%|██████████| 50/50 [00:06<00:00,  7.95it/s]
  0%|          | 0/49 [00:00<?, ?it/s]creating bow histograms for test set (pos) ...
100%|██████████| 49/49 [00:06<00:00,  7.85it/s]
testing pos samples ...
test pos sample accuracy: 0.8571428571428571
creating bow histograms for test set (neg) ...
100%|██████████| 50/50 [00:06<00:00,  7.62it/s]
testing neg samples ...
test neg sample accuracy: 0.92
```

Figure 1.2: k=15, numiter=20

```
/Users/lara/opt/anaconda3/envs/cv-lab04/bin/python /Users/lara/Desktop/ETH/courses/Computer Vison/lab04/exercise4_object_recognition_code/bow_main.py
creating codebook ...
100%|██████████| 100/100 [00:12<00:00,  8.18it/s]
number of extracted features:  10000
clustering ...
  0%|          | 0/50 [00:00<?, ?it/s]creating bow histograms (pos) ...
100%|██████████| 50/50 [00:06<00:00,  7.79it/s]
  0%|          | 0/50 [00:00<?, ?it/s]creating bow histograms (neg) ...
100%|██████████| 50/50 [00:06<00:00,  7.97it/s]
  0%|          | 0/49 [00:00<?, ?it/s]creating bow histograms for test set (pos) ...
100%|██████████| 49/49 [00:06<00:00,  7.94it/s]
testing pos samples ...
test pos sample accuracy: 0.8571428571428571
creating bow histograms for test set (neg) ...
100%|██████████| 50/50 [00:06<00:00,  7.95it/s]
testing neg samples ...
test neg sample accuracy: 0.9
```

Figure 1.3: k=15, numiter=30

Section 2

CNN-based Classifier

In this section we are asked to implement a simplified version of the VGG image classification network on CIFAR-10 dataset.

2.1 Nearest Neighbor Classification

To start with, we need to implement the network architectures in the initial function and the `forward()` function of class Vgg.
The structure of the architecture provided was the following

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 2.1: simplified VGG network architecture

Note that the first 5 blocks consist of a Conv2D followed by a ReLU and a MaxPool2D operation, while the last one implements a Linear layer followed by a ReLU, a dropout and a final Linear layer. The dropout probability I choose is 0.5, while the output size of the intermediate layer I used was 256.

```
class Vgg(nn.Module):
    def __init__(self, fc_layer=512, classes=10):
        super(Vgg, self).__init__()

        self.fc_layer = fc_layer
        self.classes = classes

        self.conv_block1 = nn.Sequential(nn.Conv2d(in_channels=3,
                                                   out_channels=64,
                                                   kernel_size=3,
                                                   stride=1,
                                                   padding=1),
                                         nn.ReLU(),
                                         nn.MaxPool2d(kernel_size=2,
                                                      stride=2))
```

```
        self.conv_block2 = nn.Sequential(nn.Conv2d(in_channels=64,
                                                   out_channels=128,
                                                   kernel_size=3,
                                                   stride=1,
                                                   padding=1),
                                         nn.ReLU(),
                                         nn.MaxPool2d(kernel_size=2,
                                                      stride=2))

        self.conv_block3 = nn.Sequential(nn.Conv2d(in_channels=128,
                                                   out_channels=256,
                                                   kernel_size=3,
                                                   stride=1,
                                                   padding=1),
                                         nn.ReLU(),
                                         nn.MaxPool2d(kernel_size=2,
                                                      stride=2))

        self.conv_block4 = nn.Sequential(nn.Conv2d(in_channels=256,
                                                   out_channels=512,
                                                   kernel_size=3,
                                                   stride=1,
                                                   padding=1),
                                         nn.ReLU(),
                                         nn.MaxPool2d(kernel_size=2,
                                                      stride=2))

        self.conv_block5 = nn.Sequential(nn.Conv2d(in_channels=512,
                                                   out_channels=512,
                                                   kernel_size=3,
                                                   stride=1,
                                                   padding=1),
                                         nn.ReLU(),
                                         nn.MaxPool2d(kernel_size=2,
                                                      stride=2))

        self.final_classifier = nn.Sequential(nn.Linear(in_features=512, out_features=256),
                                              nn.ReLU(),
                                              nn.Dropout(p=0.5),
                                              nn.Linear(in_features=256, out_features=10))

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
                m.bias.data.zero_()


    def forward(self, x):
        score = None
        ris = x.clone()
        s_stacked = nn.Sequential(self.conv_block1,
                                  self.conv_block2,
                                  self.conv_block3,
                                  self.conv_block4,
                                  self.conv_block5)
        ris = s_stacked(ris)
        ris = ris[:,:,0,0] # The input to the linear layer must be a 2D tensor
        score = self.final_classifier(ris)

        return score
```

Note that before feeding the linear layer, it is necessary to convert the output of the 5^{th} Convolutional layer to a 2D tensor.

2.2 Training and testing

The hyperparameters I used were the default ones.

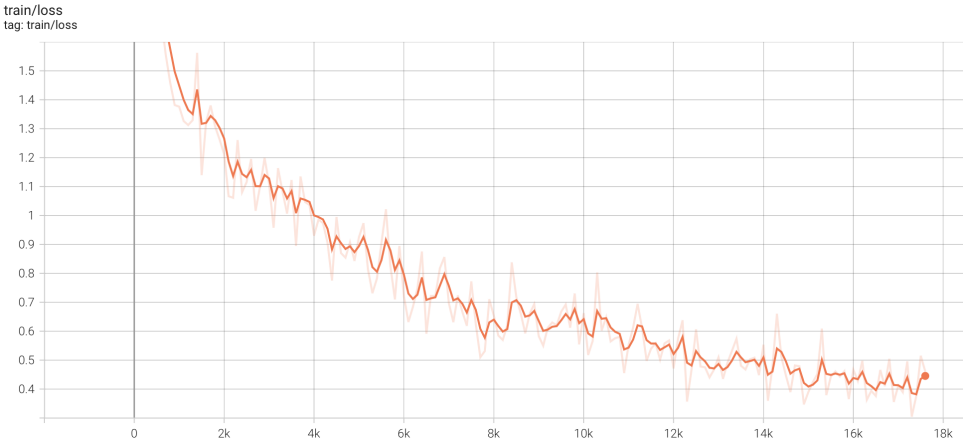

Figure 2.2: Tensorboard Plots: train/loss

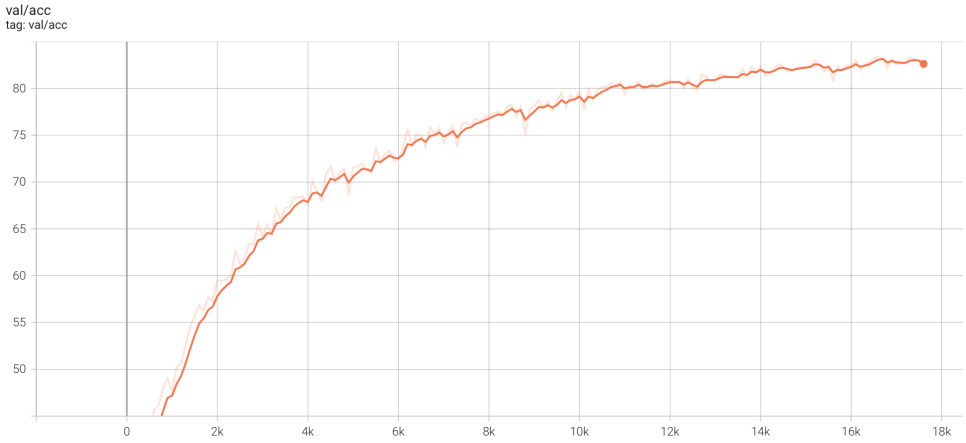

Figure 2.3: Tensorboard Plots: acc

To conclude, the final results were:

- best model saved loss : 0.4653
- best model saved validation accuracy: 83.44
- test accuracy on CIFAR-10 test set: 81.71