Report

2. Bag-of-words Classifier

2.1 Local Feature Extraction

2.1.1 Feature Detection – Feature Points on A Grid

According to the task requirement, we need first remove 8 pixels on both sides in each dimension of the image and then obtain grid points such that they are equally spaced in the resulting remaining graph. The code is as followed:

```
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]
    """
    # vPoints = None  # numpy array, [nPointsX*nPointsY, 2]

    # todo
    h = img.shape[0]
    w = img.shape[1]
    dx = (w-border-1-border)/(nPointsX-1)
    dy = (h-border-1-border)/(nPointsY-1)
    vPoints = np.zeros((nPointsX*nPointsY, 2))

    for j in range(nPointsY):
        for i in range(nPointsX):
            vPoints[j*nPointsX+i][0] = border + i*dx
            vPoints[j*nPointsX+i][1] = border + j*dy

    return vPoints
```

2.1.2 Feature Description – Histogram of Oriented Gradients

For each feature point (the grid point), we need to compute a 128-d vector as its descriptor. Note that the coordinates of the grid point are not necessarily integers so we need to first assign them to the nearest integer to make them match the pixel indexes:

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

Then for each grid point, we consider 16 neighbour cells surrounding it. For each cell, we need to compute a 8-d vector to store the gradient histogram of that cell. More specifically, we separate the gradient direction [0, 2pi) in 8 intervals of equal length: [0, pi/4), [pi/4, pi/2), ... [7/4pi, 2pi), if the gradient direction belongs to k^{th} (0-index) interval [k×pi/4, (k+1)×pi/4), then we vote k^{th} interval. To take the magnitude of the gradient into account, we use the magnitude of the corresponding gradient as the weight when voting. The code is as followed:

```
CurrentCell_histogram = np.zeros(nBins)
for j in range(h):
    for i in range(w):
        current_pixel_grad_x = grad_x[start_y+j][start_x+i]
        current_pixel_grad_y = grad_y[start_y+j][start_x+i]
        current_pixel_grad_angle = np.arctan2(current_pixel_grad_y,current_pixel_grad_x)
        current_pixel_total_grad = np.sqrt(np.power(current_pixel_grad_x,2)+np.power
        (current_pixel_grad_y,2)) # weighted vote
        index = current_pixel_grad_angle/angleInterval
        if index < 0:
            index = np.floor(index) + nBins
        else:
            index = np.floor(index)
        index = int(index)
        CurrentCell_histogram[index] += current_pixel_total_grad

for i in range(nBins):
    desc.append(CurrentCell_histogram[i])
```

After we computed the histogram vector for each cell and append them one after another, we obtain the 128-d vector desc. We normalize this vector to make its 2-norm is equal to 1:

```
# normalize desc
desc = list(desc/np.linalg.norm(np.array(desc)))
```

We use the same procedure to compute all the normalized 128-d vector for all the grid points, then we get our full descriptor.

2.2 Codebook Construction

To construct the codebook, we extract all the grid points of all the training images and obtain their descriptors (128-d vectors). After we get all the descriptors, we use k-means algorithm to cluster them. Each cluster center represents a “word”. The code for collecting all the descriptors is as followed:

```
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, nPoint(variable) img: Mat
    current_img_descriptors = descriptors_hog(img, vPoints, cellWidth, cellHeight) # return np.array
    vFeatures.append(list(current_img_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))
```

2.3 Bag-of-words Vector Encoding

2.3.1 Bag-of-words Histogram

After constructing the codebook, we can compute the Bag-of-words histogram for a given image. More precisely, we can assign the descriptors of each grid point of that image to the closest cluster center. Then we can count the number of the grid points assigned to each cluster center to build the Bag-of-words histogram for that image. We exploit the given findnn function to assign the descriptors:

```
# todo
Idx, Dist = findnn(vFeatures,vCenters) # Idx, Dist are M-D vectors
Idx = np.asarray(Idx)
histo = np.zeros(vCenters.shape[0])

for i in range(vCenters.shape[0]):
    histo[i] = np.count_nonzero(Idx == i)
```

2.3.2 Processing A Directory with Training Examples

We follow the same procedure to build Bag-of-words histogram for each training image and put them all together to build a N × k matrix, where N is the number of training image and k is the number of clusters. The code is as followed:

```
# 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]

    # todo
    vPoints = grid_points(img, nPointsX, nPointsY, border)
    current_img_descriptors = descriptors_hog(img, vPoints, cellWidth, cellHeight) # return np.array
    vFeatures = current_img_descriptors
    histo = bow_histogram(vFeatures, vCenters)
    vBoW.append(list(histo))


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

2.4 Nearest Neighbour Classification

To classify a test image, we compute its bag-of-words histogram, and assign it to the category of its nearest neighbour training histogram. Again, we exploit the findnn function to compute the minimum distance between the test image's histogram and the histograms in both category (positive and negative) and compare the 2 values to classify the image. The code is as followed:

```
# Find the nearest neighbor in the positive and negative sets and decide based on this neighbor
# todo
IdxPos, DistPos = findnn(histogram,vBoWPos)
IdxNeg, DistNeg = findnn(histogram,vBoWNeg)

if (DistPos < DistNeg):
    sLabel = 1
else:
    sLabel = 0
return sLabel
```

Run the whole python file and we can get a result like this:

```
creating codebook ...
100%|██████████| 100/100 [00:23<00:00,  4.21it/s
number of extracted features:  10000
clustering ...
creating bow histograms (pos) ...
100%|██████████| 50/50 [00:12<00:00,  4.13it/s
creating bow histograms (neg) ...
100%|██████████| 50/50 [00:12<00:00,  4.14it/s
creating bow histograms for test set (pos) ...
100%|██████████| 49/49 [00:12<00:00,  3.93it/s
testing pos samples ...
test pos sample accuracy: 0.9183673469387755
creating bow histograms for test set (neg) ...
100%|██████████| 50/50 [00:12<00:00,  3.91it/s
]
testing neg samples ...
test neg sample accuracy: 1.0
```

The accuracy we obtained is fairly good. Note that the result would be slightly different at each run because of the random initialization of k-means clustering.

3. CNN-based Classifier

3.1 A Simplified Version of VGG Network

According to the given simplified VGG network architecture, define the model of the network as followed:

```
self.model = nn.Sequential(
            nn.Conv2d(in_channels=3,out_channels=64,kernel_size=3,padding=1), #output[bs, 64, 32, 32]
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,stride=2, padding=0), #output[bs, 64, 16, 16]

            nn.Conv2d(in_channels=64,out_channels=128,kernel_size=3,padding=1), #output[bs, 128, 16, 16]
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,stride=2, padding=0), #output[bs, 128, 8, 8]

            nn.Conv2d(in_channels=128,out_channels=256,kernel_size=3,padding=1), #output[bs, 256, 8, 8]
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,stride=2, padding=0), #output[bs, 256, 4, 4]

            nn.Conv2d(in_channels=256,out_channels=512,kernel_size=3,padding=1), #output[bs, 512, 4, 4]
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,stride=2, padding=0), #output[bs, 512, 2, 2]

            nn.Conv2d(in_channels=512,out_channels=512,kernel_size=3,padding=1), #output[bs, 512, 4, 4]
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,stride=2, padding=0), #output[bs, 512, 1, 1]

            nn.Flatten(),
            nn.Linear(512,self.fc_layer),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(self.fc_layer,self.classes)
            )
```

Note that we need to introduce nn.Flatten layer before the linear layer to make the dimension fit.

Use the model defined as forward pass:

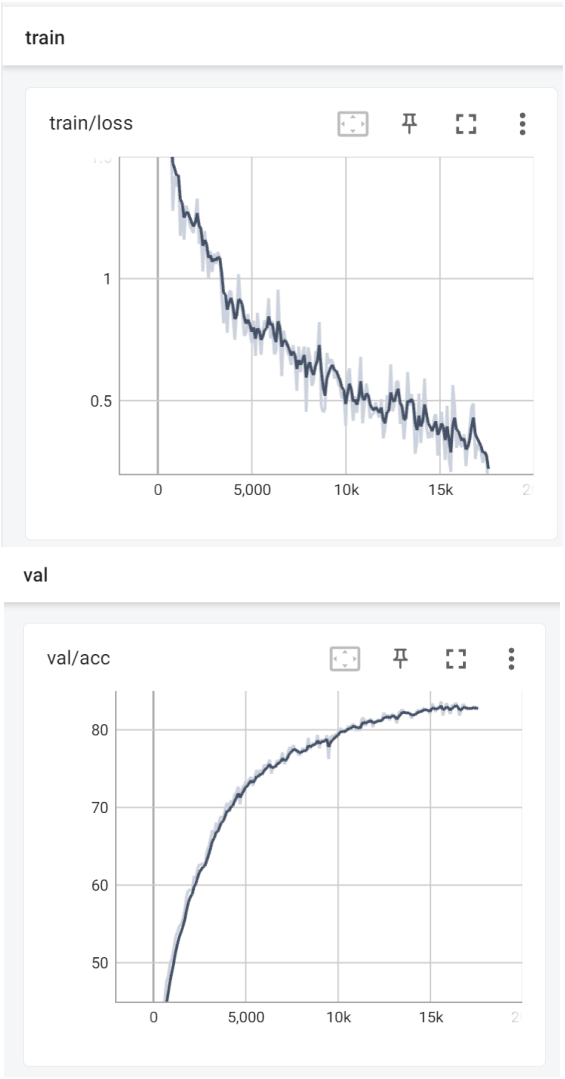
```
def forward(self, x):
    """
    :param x: input image batch tensor, [bs, 3, 32, 32]
    :return: score: predicted score for each class (10 classes in total), [bs, 10]
    """
    score = None
    # todo
    score = self.model(x)

    return score
```

3.2 Training and Testing

3.2.1 Training

Use the simplified VGG network to train, we get the following training loss and validation accuracy curves:



3.2.2 Testing

After training the model, we use the test data to verify the accuracy. We get the following result:

```
(base) D:\Studies\Courses\Computer Vision\Homework\lab04 Object Recognition\CV_2022_exercise_4\CV_2022_exercise_4\ex
ercise4_object_recognition_code>python test_cifar10_vgg.py
[INFO] test set loaded, 10000 samples in total.
79it [00:06, 12.16it/s]
test accuracy: 82.04
```

The test accuracy is 0.8204, which exceeds 0.75, as required.