Object Recognition

Computer Vision Lab 04 Report
HS2023

Christopher Zanoli

czanoli@ethz.ch
Student Number: 23-942-394

Abstract

The report is divided into two main sections: **Bag-of-Words Classifier** and **CNN-based Classifier**. Within each section, I provide a comprehensive analysis of the implementation decisions and clarify the results both at the level of theoretical foundations and empirical observations.

1 Bag-of-Words Classifier

The Bag-of-Words (BoW) Classifier serves as a simplified representation of the visual features encountered within an image, by quantifying the presence of certain visual "words" without considering their spatial relationship. The purpose of BoW is twofold:

- It facilitates image classification by converting high-dimensional image data into a manageable form, i.e. histogram signatures, that can be efficiently analyzed;
- It supports object recognition within images, as the presence of certain visual words can indicate the presence of specific objects, despite variations in scale (for example when employing SIFT).

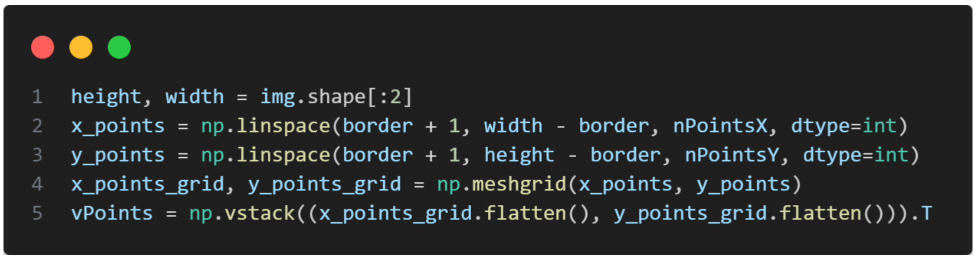
The following are the main steps of Bag-of-Words Classifier:

1.1 Local Feature Extraction

1.1.1 Feature Detection - feature points on a grid

The `grid_points()` function is designed to generate a set of points that are evenly distributed across the input image, while excluding a border region to avoid edge effects. This evenly spaced grid of points can then be used for sampling features.

Figure 1 shows the implementation:



```python
height, width = img.shape[:2]
x_points = np.linspace(border + 1, width - border, nPointsX, dtype=int)
y_points = np.linspace(border + 1, height - border, nPointsY, dtype=int)
x_points_grid, y_points_grid = np.meshgrid(x_points, y_points)
vPoints = np.vstack((x_points_grid.flatten(), y_points_grid.flatten())).T
```

Figure 1: grid_points() implementation.

The function constructs two arrays: one for the x-coordinates and one for the y-coordinates of the grid points. It achieves this by using `np.linspace()`, ensuring that the points are evenly distributed and that the border is excluded by starting the linspace from the border and ending it at `width - border` for the x-axis and at `height - border` for the y-axis. The `dtype=int` ensures the coordinates are integers, which is necessary for correctly indexing pixels in the input image. After that, it forms the grid by combining the x-coordinates and y-coordinates into a single array of point coordinates by meshing them together using `np.meshgrid()` and then flattening and transposing the arrays to obtain a list of coordinate pairs. The resulting vPoints, is a 2D numpy array of shape `[nPointsX*nPointsY, 2]`, where each row corresponds to the `[x, y]` coordinates of a point on the grid.

Figure 2 shows an example of the computed grid overlaid on an input image:

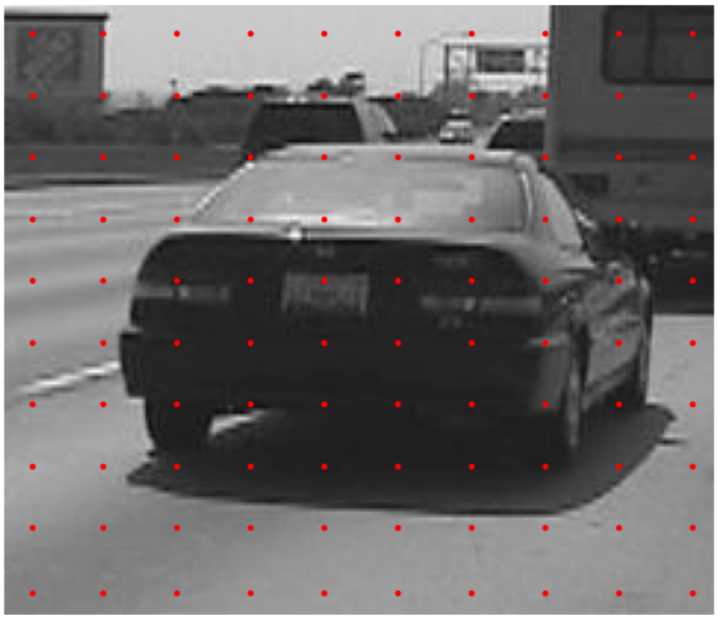


Figure 2: Grid on image 5 of *cars-training-pos*.

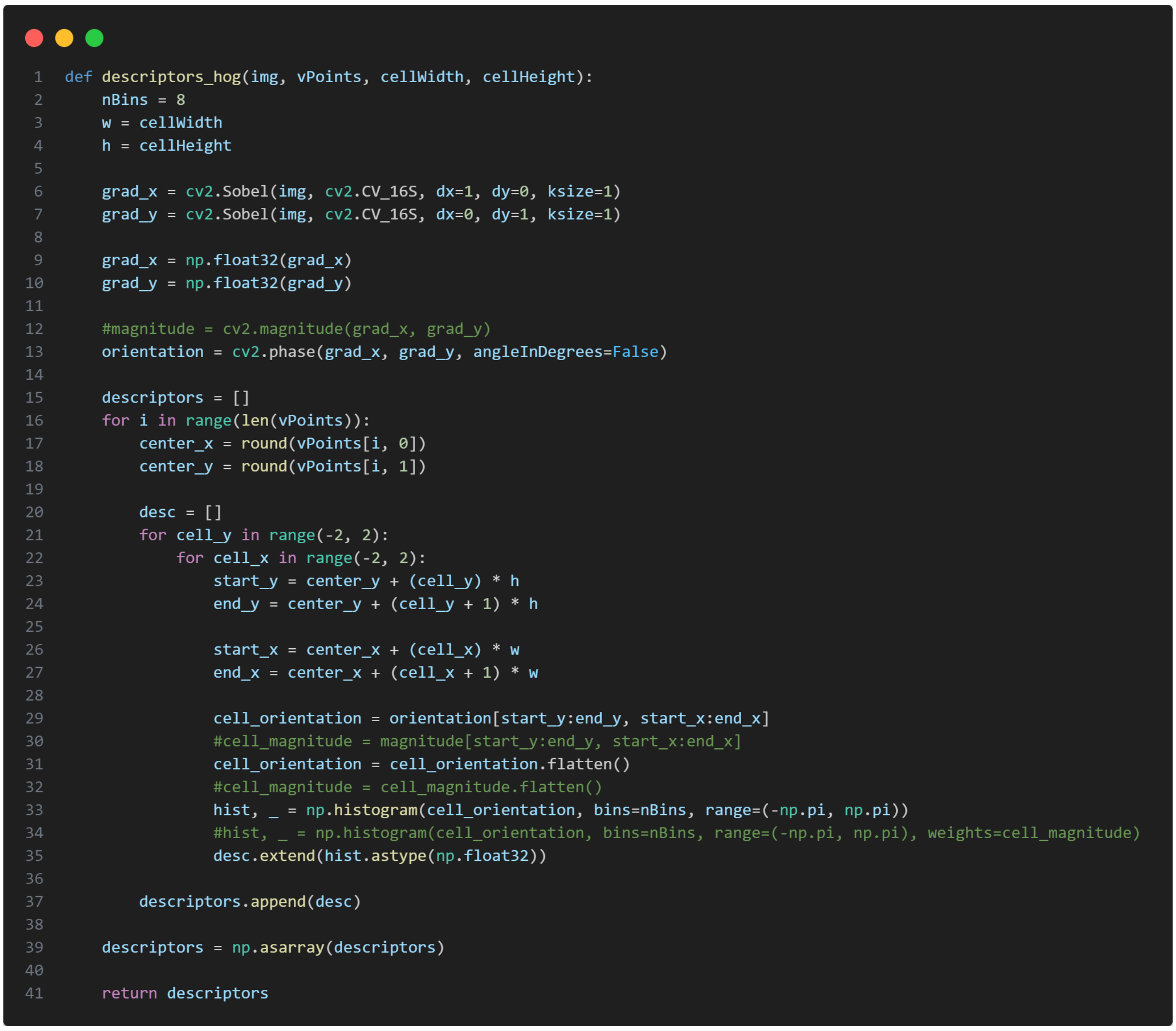
1.1.2 Feature Description - histogram of oriented gradients

The `descriptors_hog()` function implements the Histogram of Oriented Gradients (HOG) descriptor, which is a technique that counts occurrences of gradient orientation in localized cells of an input image.

Figure 3 shows the implementation.

The gradients in both the x and y directions are computed using the *Sobel* operator. The gradients are computed with a kernel size of 1, in order to preserve finer details (i.e. no Gaussian smoothing applied), and then are converted to a `np.float32` type, which is necessary for further calculations involving the gradients. The orientation of the gradient at each pixel is computed using the `cv2.phase()` function, which calculates the angle of the gradient vector.
A loop is run for each of the points in `vPoints`, i.e. the centers of the cells on which the HOG descriptor will be computed. Each `vPoint` will have an associated descriptor that captures the distribution of gradient orientations around that point.
For each center point, a `4x4` cell grid is considered, and for each cell, the histogram of gradient orientations is computed using the `np.histogram()` function. The range for the histogram is from `-np.pi` to `np.pi`, covering all possible angles. The variable `nBins` is set to 8, which specifies the number of bins in the histogram of gradients. Therefore, since we have a `4x4` grid of cells, and each cell contributes 8 elements to the histogram, the total number of elements in the descriptor for each point would be: 4 x 4 x 8 = 128 elements.
The histograms from the `4x4` grid are flattened and concatenated to form the final feature vector for each grid point. This vector is then added to the list of descriptors, and the latter is finally converted to a numpy array, whose final shape is `[nPointsX*nPointsY, 128]`.

Note: *As specifically requested in the handout, the histogram does not use the gradient magnitude as a weight, which is a simplification and differs from the standard HOG implementation. Therefore, the lines of code related to standard HOG implementation that leverages magnitude are commented out.*



```python
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)

    grad_x = np.float32(grad_x)
    grad_y = np.float32(grad_y)

    #magnitude = cv2.magnitude(grad_x, grad_y)
    orientation = cv2.phase(grad_x, grad_y, angleInDegrees=False)

    descriptors = []
    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

                cell_orientation = orientation[start_y:end_y, start_x:end_x]
                #cell_magnitude = magnitude[start_y:end_y, start_x:end_x]
                cell_orientation = cell_orientation.flatten()
                #cell_magnitude = cell_magnitude.flatten()
                hist, _ = np.histogram(cell_orientation, bins=nBins, range=(-np.pi, np.pi))
                #hist, _ = np.histogram(cell_orientation, bins=nBins, range=(-np.pi, np.pi), weights=cell_magnitude)
                desc.extend(hist.astype(np.float32))

        descriptors.append(desc)

    descriptors = np.asarray(descriptors)

    return descriptors
```

Figure 3: descriptors_hog() implementation.

1.2 Codebook Construction

The `create_codebook()` function generates a visual vocabulary by clustering feature descriptors from the set of training images and quantizes the feature space into `k` distinct visual words. This is a crucial step for further classification.

Figure 4 shows the implementation.

The function first aggregates all the images (both positive and negative) from the training directories. For each image, the following steps are executed:

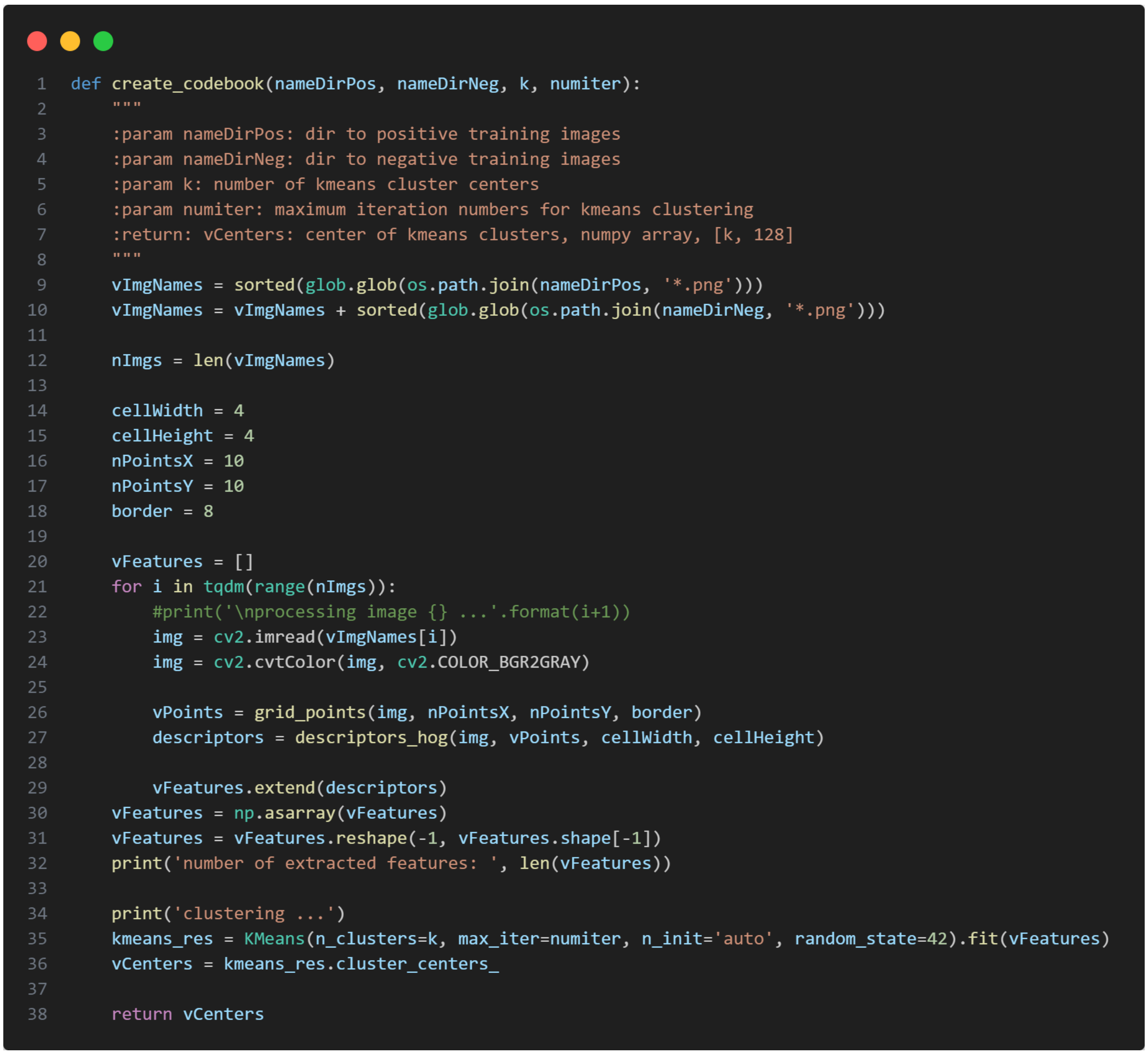
- The image is read and converted to a gray-scale format;
- A set of pre-defined grid points is computed over the image using the previously described `grid_points()` function;
- For each grid point, a HOG descriptor is computed using the previously described `descriptors_hog()` function.

All HOG descriptors from all images are collected into a single array. Finally, K-means clustering is applied to this array of descriptors to find `k` cluster centers, which represent the most prominent patterns in the feature space across all training images.
The output is `vCenters`, i.e. the clusters centers, represented as an array of shape `[k, 128]`, where each row corresponds to a cluster center in the descriptor space.

Note: The results that will be presented in section 1.6 were obtained based on a specific value of `k`, specifically `k=190`. This value was determined through a hyperparameter tuning process where I experimented with `k` ranging from 10 to 250 in increments of 10, totaling 25 distinct values. For the sake of simplicity and ease of implementation, the number of iterations was held constant at `numiter=300`.

Within the submitted code, there are comments that deal with the hyperparameter tuning process.

In section 1.6, I will show and discuss a graph that depicts the variation in accuracy across the spectrum of tested k values.



```python
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 = []
    for i in tqdm(range(nImgs)):
        #print('\nprocessing image {} ...'.format(i+1))
        img = cv2.imread(vImgNames[i])
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        vPoints = grid_points(img, nPointsX, nPointsY, border)
        descriptors = descriptors_hog(img, vPoints, cellWidth, cellHeight)

        vFeatures.extend(descriptors)
    vFeatures = np.asarray(vFeatures)
    vFeatures = vFeatures.reshape(-1, vFeatures.shape[-1])
    print('number of extracted features: ', len(vFeatures))

    print('clustering ...')
    kmeans_res = KMeans(n_clusters=k, max_iter=numiter, n_init='auto', random_state=42).fit(vFeatures)
    vCenters = kmeans_res.cluster_centers_

    return vCenters
```

Figure 4: create_codebook() implementation.

1.3 Bag-of-Words Vector Encoding

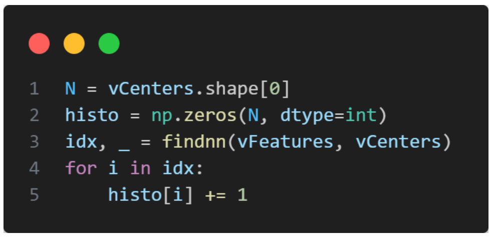
1.4 Bag-of-Words histogram

The `bow_histogram()` function translates the visual content of an image into a histogram based on the previously defined vocabulary of visual words, allowing for a quantitative comparison between images based on their visual features.

Figure 5 shows the implementation.

Firstly, the number of cluster centers is determined by the first dimension of `vCenters`, and the array `histo` is initialized accordingly. After that, the function `findnn()` (already provided) is called to find the nearest neighbors of the feature vectors in `vFeatures` among the cluster centers in `vCenters`. It returns

indices of the closest cluster centers for each feature.
The function then iterates over these indices `idx` and increments the corresponding position in `histo` for each index. This counts how many times each cluster center is the nearest neighbor to the feature vectors. The function returns the `histo` vector, which now contains the complete BoW histogram for the input image features. Each element of this histogram represents the frequency of its corresponding visual word within the image.



```python
N = vCenters.shape[0]
histo = np.zeros(N, dtype=int)
idx, _ = findnn(vFeatures, vCenters)
for i in idx:
    histo[i] += 1
```

Figure 5: bow_histogram() implementation.

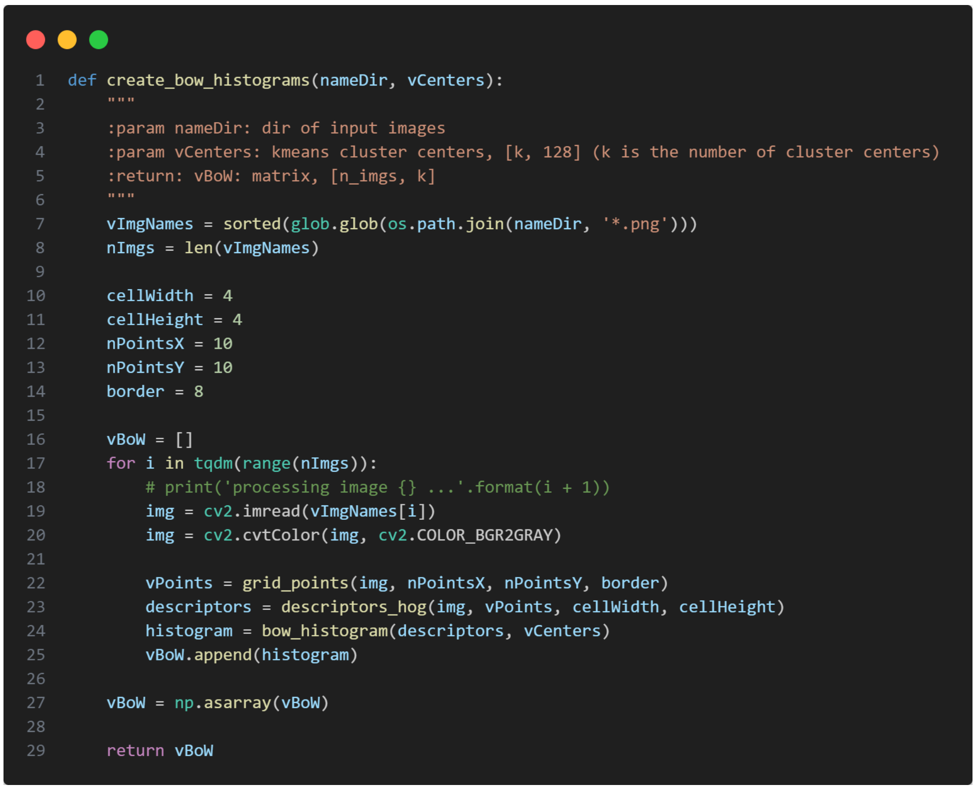
At this point, it is necessary to complete the `create_bow_histogram()` function, which is needed to read all the training images and calculate a bag-of-words histogram for each.

Figure 6 shows the implementation.

The function iterates over each image name, reads the image in grayscale, and then:

- It computes a set of grid points on the image using the previously described `grid_points()` function discussed in section 1.1.1;
- For each grid point, it calculates the HOG descriptor using the `descriptors_hog()` function discussed in section 1.1.2;
- For the descriptors obtained from a single image, it computes the BoW histogram using the `bow_histogram()` function discussed in section 1.4. Each histogram is appended to vBoW.

After all images are processed, vBoW is converted to a numpy array and the final shape is `[n_imgs*k]`.



```python
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')))
    nImgs = len(vImgNames)

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

    vBoW = []
    for i in tqdm(range(nImgs)):
        # print('processing image {} ...'.format(i + 1))
        img = cv2.imread(vImgNames[i])
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

    vBoW = np.asarray(vBoW)

    return vBoW
```

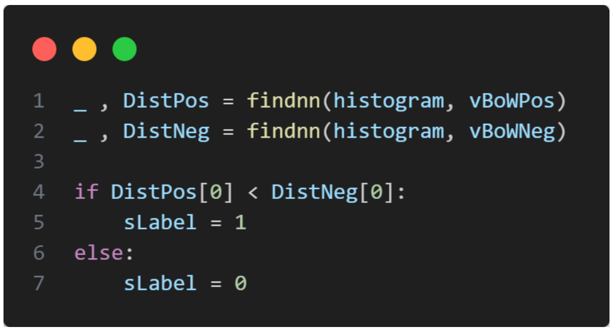
Figure 6: create_bow_histogram() implementation.

1.5 Nearest Neighbor Classification

The already provided `findnn()` function computes the distance of the test histogram to each of the histograms in the training sets (both positive and negative), returning a list of distances: `DistPos` for positive and `DistNeg` for negative.
Since the goal is to find the nearest neighbor within the positive and negative training sets, we are only interested in the smallest distances from these lists. Hence, with `DistPos[0]` and `DistNeg[0]`, we access their first element, which contain the smallest distance from the test histogram to the positive histograms and the smallest distance to the negative histograms, respectively.
After that, the decision rule is implemented as follows:

- If the test image histogram is closer to the positive histograms, it is classified as a positive image;
- If the test image histogram is closer to the negative histograms, it is classified as a negative image.

Figure 7 shows the implementation.



```python
_ , DistPos = findnn(histogram, vBoWPos)
_ , DistNeg = findnn(histogram, vBoWNeg)

if DistPos[0] < DistNeg[0]:
    sLabel = 1
else:
    sLabel = 0
```

Figure 7: bow_recognition_nearest() implementation.

1.6 Results

This section presents the results obtained. As anticipated in section 1.2, I decided to perform tuning of the `KMeans()` hyperparameter `k`. Figure 8 shows the values of accuracies based on the value of `k`. In total, I tested 25 different values from 10 to 250 (step=10).

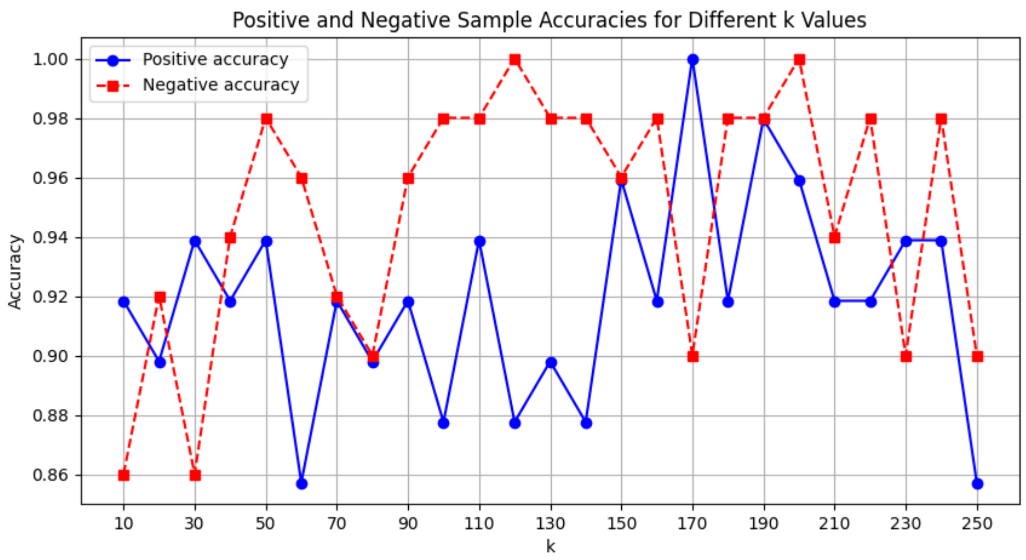
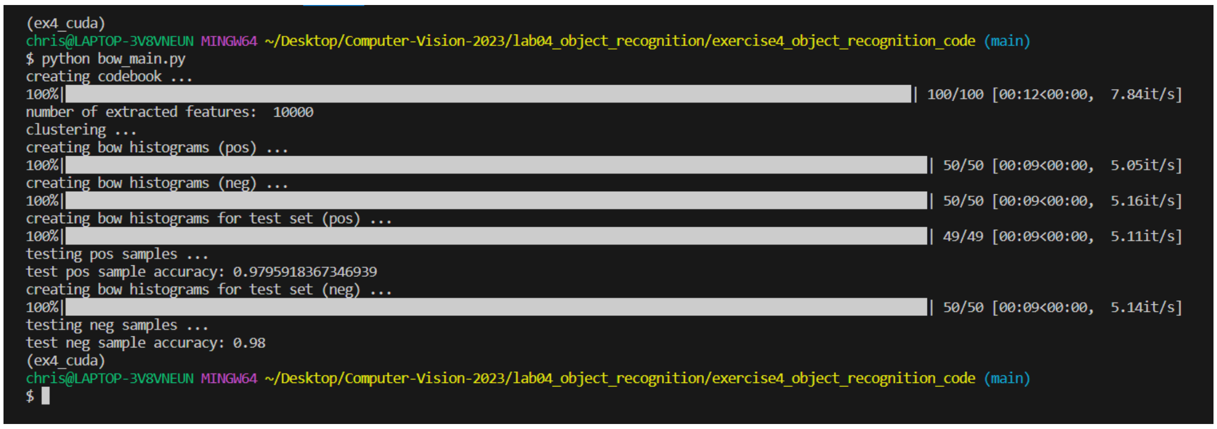


Figure 8: Variation of classification accuracies with k.

It can be seen that, in general, any value of `k` allows satisfactory accuracy values to be obtained, both for test images related to positive cases and for test images related to negative cases. It is interesting to note that, for `k` between 80 and 150 (extremes excluded), the discrepancy between the accuracy obtained on positive cases and the accuracy obtained on negative cases is more pronounced than for other values of `k`. Since there are 2 accuracies, I chose the value of `k` that would allow for two accuracies to be as similar and high as possible: `k=190` meets this requirement, as it allows for both positive and negative cases to obtain an accuracy of about 98%.

As expressly requested by the handout, I report the figure 9, containing the logs printed on the command line interface after the execution of `bow_main.py` with `k=190`.



```
(ex4_cuda)
chris@LAPTOP-3V8VNEUN MINGW64 ~/Desktop/Computer-Vision-2023/lab04_object_recognition/exercise4_object_recognition_code (main)
$ python bow_main.py
creating codebook ...
100%|██████████████████████████████████████████████████| 100/100 [00:12<00:00,  7.84it/s]
number of extracted features:  10000
clustering ...
creating bow histograms (pos) ...
100%|██████████████████████████████████████████████████| 50/50 [00:09<00:00,  5.05it/s]
creating bow histograms (neg) ...
100%|██████████████████████████████████████████████████| 50/50 [00:09<00:00,  5.16it/s]
creating bow histograms for test set (pos) ...
100%|██████████████████████████████████████████████████| 49/49 [00:09<00:00,  5.11it/s]
testing pos samples ...
test pos sample accuracy: 0.9795918367346939
creating bow histograms for test set (neg) ...
100%|██████████████████████████████████████████████████| 50/50 [00:09<00:00,  5.14it/s]
testing neg samples ...
test neg sample accuracy: 0.98
(ex4_cuda)
chris@LAPTOP-3V8VNEUN MINGW64 ~/Desktop/Computer-Vision-2023/lab04_object_recognition/exercise4_object_recognition_code (main)
$ █
```

Figure 9: bow_main.py logs.

2 CNN-based Classifier

2.1 VGG Network

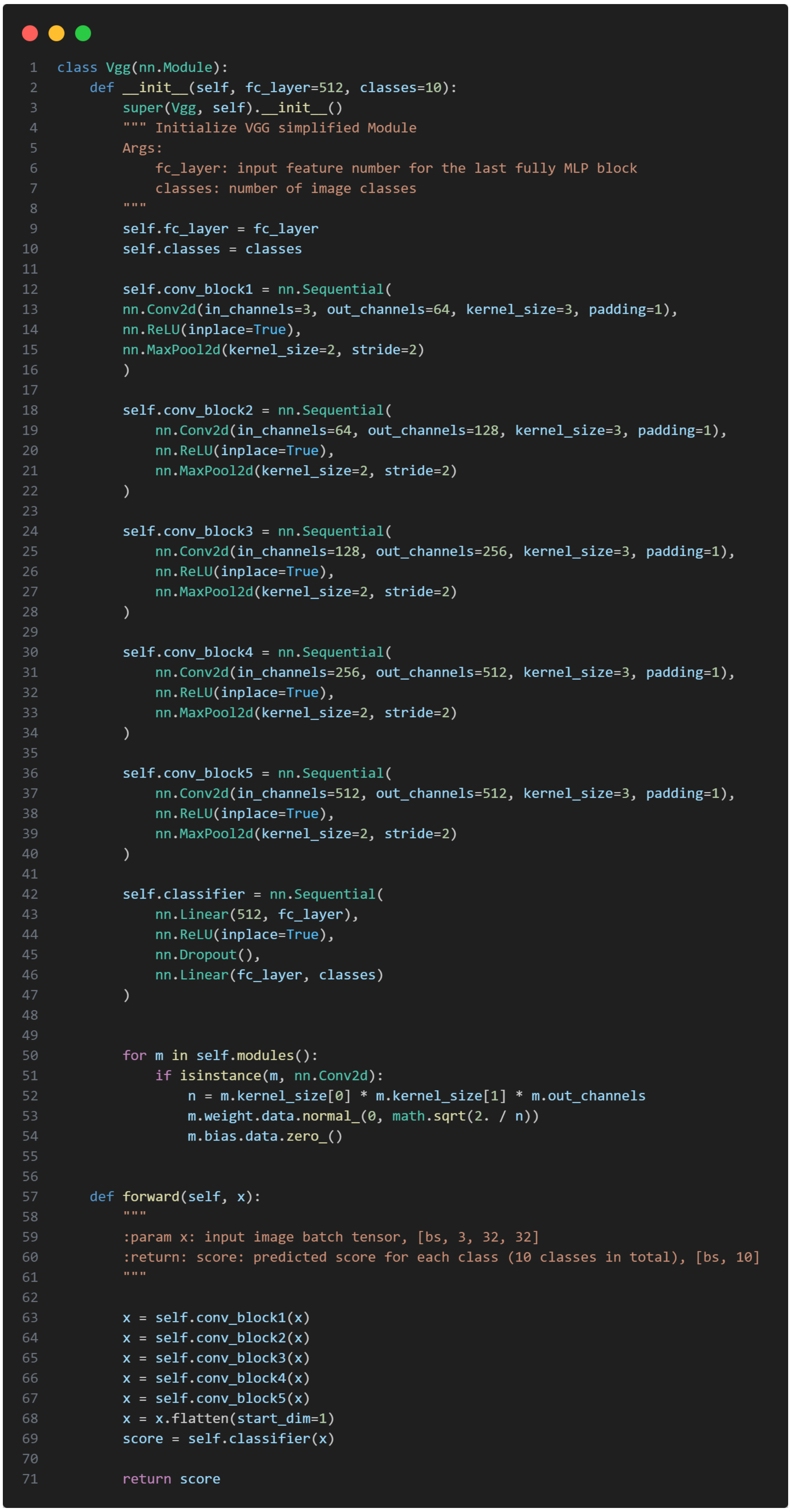
According to [1] The VGG architecture aims to increase the depth of convolutional neural networks. To do so, it exploits small 3x3 filters and only two additional components: pooling layers and fully connected layers.

The variant of the VGG network, implemented in the `Vgg()` class shown in figure 10, adapts the core ideas of the original VGG network for image classification tasks. In particular, this implementation is simplified compared to the full VGG networks (e.g., VGG-16 or VGG-19), which contain more convolutional layers and larger input image sizes.This version is tailored for smaller images, hence it is suitable for tasks such as CIFAR-10 image classification, where input images are relatively small (32x32).

The initialization parameters are `fc_layer`, i.e. the number of neurons in the fully-connected layer before the output layer, and `classes`, i.e., the number of classes to predict.
The network architecture comprises several blocks of convolutional (`nn.Conv2d()`) layers followed by Rectified Linear Unit (`nn.ReLU()`) activations and Max Pooling (`nn.MaxPool2d()`) operations.The sequential blocks progressively downsample the input image while increasing the feature map depth.

After the series of convolutional blocks, the image features are flattened, and a sequence of fully connected layers (`nn.Linear()`) interspersed with ReLU activation and Dropout (`nn.Dropout()`) for regularization is applied to classify the image into one of the classes.

The `forward()` method implements the *forward pass*: the input tensor `x` is passed through all five convolutional blocks, flattened, and then passed through the classifier to produce the final class scores.



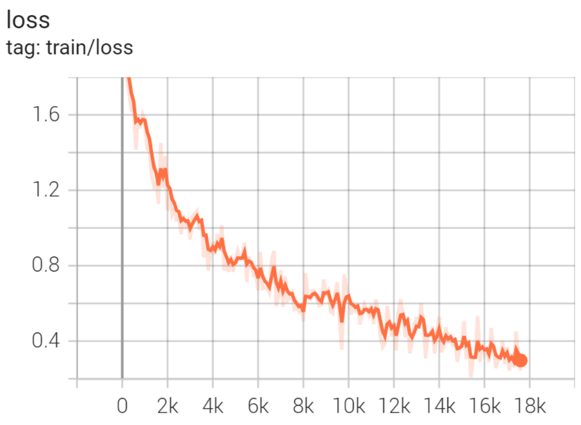
```python
class Vgg(nn.Module):
    def __init__(self, fc_layer=512, classes=10):
        super(Vgg, self).__init__()
        """ Initialize VGG simplified Module
        Args:
            fc_layer: input feature number for the last fully MLP block
            classes: number of image classes
        """
        self.fc_layer = fc_layer
        self.classes = classes

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

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

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

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

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

        self.classifier = nn.Sequential(
            nn.Linear(512, fc_layer),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(fc_layer, classes)
        )


        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):
        """
        :param x: input image batch tensor, [bs, 3, 32, 32]
        :return: score: predicted score for each class (10 classes in total), [bs, 10]
        """

        x = self.conv_block1(x)
        x = self.conv_block2(x)
        x = self.conv_block3(x)
        x = self.conv_block4(x)
        x = self.conv_block5(x)
        x = x.flatten(start_dim=1)
        score = self.classifier(x)

        return score
```

Figure 10: Vgg() class implementation

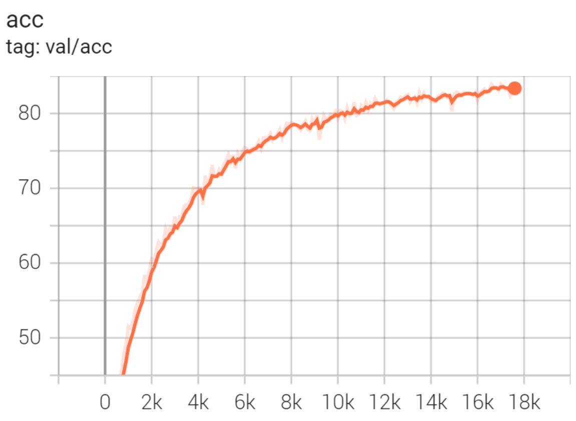
2.2 Results

This section shows the results obtained. For the sake of simplicity and ease of implementation, I decided to use the default hyperparameters set. The training was done by taking advantage of my laptop's dedicated `GeForce GTX 1650 Max-Q` GPU, and the time required for training was about 44 minutes and 6 seconds.

As specifically requested in the handout I report the two `TensorBoard` graphs, showing the training loss and validation accuracy respectively.



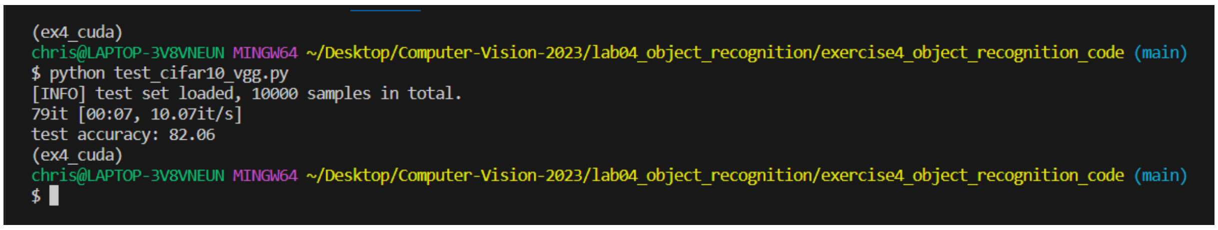
(a) Training loss.



(b) Validation accuracy.

The final accuracy obtained on the test set is: **82.06%**.

Figure 12 shows the logs printed on command line interface after the execution of `test_cifar10_vgg.py`.



```
(ex4_cuda)
chris@LAPTOP-3V8VNEUN MINGW64 ~/Desktop/Computer-Vision-2023/lab04_object_recognition/exercise4_object_recognition_code (main)
$ python test_cifar10_vgg.py
[INFO] test set loaded, 10000 samples in total.
79it [00:07, 10.07it/s]
test accuracy: 82.06
(ex4_cuda)
chris@LAPTOP-3V8VNEUN MINGW64 ~/Desktop/Computer-Vision-2023/lab04_object_recognition/exercise4_object_recognition_code (main)
$
```

Figure 12: test_cifar10_vgg.py logs.

References

[1] K Simonyan and A Zisserman. Very deep convolutional networks for large-scale image recognition. pages 1–14. Computational and Biological Learning Society, 2015.