Computer Vision Assignment 5

Saurabh Vaishampayan
svaishampaya@student.ethz.ch

December 2021

1 Bag of Words Classifier

We briefly describe the implementation for each of the subtasks for the question 1: Implementing a bag of words classifier. Then we give results and observations for the task.

1.1 Feature points on a grid

This is implemented in the grid_points() function. We have to divide each image into a nPointsX$\times$ nPointsY grid, where nPointsX = nPointsY = 10 We leave out border number of points from the beginning and end of each axis. The spacing between adjacent grid points is equal to

$$\delta\text{x} = \text{floor}\frac{(w - 2 \times \text{border} - 1)}{\text{nPointsX} - 1}$$

$$\delta\text{y} = \text{floor}\frac{(h - 2 \times \text{border} - 1)}{\text{nPointsY} - 1}$$

We create indices for each axis using np.arange(), and get all sets of 2D indices by reshaping the meshgrid formed by the indices along each dimension. The list of 2D indices is returned by the function.

1.2 Histogram of oriented gradients

Now our task is to create a feature descriptor (in terms of gradient orientations) for each image. This is implemented in descriptors_hog() function. We briefly describe the steps below:

1. For each image we have the set of 10×10 grid point indices. For each of the gridpoints we consider a neighbourhood of pixels (given by a rectangular patch centered at the grid point of interest). The rectangular patch is divided into 16 cells, each cell itself consisting of 16 points.

2. For each of the 16 cells, we have to compute the histogram of oriented gradients. The final feature vector for each gridpoint is given by concatenation of histogram for each of the 16 cells.

3. For each of the $4 \times 4 = 16$ points in each cell, we have to compute the orientation of the gradient. This is given by an angle in the range $[0, 2\pi)$. This can be computed using np.arctan2($\nabla_x I, \nabla_y I$). np.arctan2() gives an angle in range $[-\pi, \pi)$ and this must be unwrapped by performing modulo w.r.t 2π. However, we are only interested in orientation being in one of 8 equidistant directions. So we must round the phase to the nearest $\pi/4$ value. Also, due to circular nature of phase, phases rounded to 2π and 0 are equivalent and this must be taken care of.

4. The above is performed for each of the 16 points in the cell, and we need to obtain the histogram representing the distribution of the directions over this set of 16 points, which is an 8 dimensional vector. This is done for each cell, and there are 16 cells in total, thus each gridpoint has a 128 d descriptor.

5. The oriented gradient descriptor is stacked for all the gridpoints, giving us a 100×128 descriptor for each image, since there are a total of 100 gridpoint locations.

1.3 Codebook construction

This is implemented in the create_codebook() function.

Now we have a descriptor for each image in the training dataset. We have $N = 100$ training images. Our task is to cluster the data by using k means. The clustering takes place in the 128 dimensional vector space. We have a set of $N_p = N * \text{nPointsX} * \text{nPointsY} = 100 * 100$ data points, each of dimension 128. We have to cluster this into k centers, and this part was already given to us using the KMeans() function from sklearn. One important point to note is that the default initialisation for KMeans() is not random, rather it is chosen internally depending on the data by scikit-learn. Finally, at the end of the function, we have the 128 cooordinates for k cluster centers, which is returned by the function.

1.4 Bag of words histogram

The following is implemented in the functions bow_histogram() and create_bow_histograms(). We also make use of the findnn() function already provided to us to find the nearest point among a set for a given point. In the above section, we obtained the k means cluster centers. Each of these k centers are "words". Our task is to convert each image into a bag of words descriptor from the oriented histogram descriptor we computed earlier. Each image has a set of $\text{nPointsX} * \text{nPointsY}$ points, with each point being of dimension $d = 128$. For each of these points, we assign a "word", given by the nearest k-means cluster center. So for each image, we now have a vector of length $\text{nPointsX} * \text{nPointsY}$, where each entry represents a "word" or the nearest k-means cluster center. Now for this, we create a histogram with k bins, where k is the nuber of cluster centers. Thus we obtain a k dimensional vector for each image that represents a histogram of number of occurences of each "word" in the image.

We perform this for each image in the training dataset. We have $N = 100$ training images, and hence we get N histograms, each of size k. We stack these as a $N \times k$ numpy array, which will be used for classification. Remember that each row of this array represents the Bag of Words Histogram for that particular image.

1.5 Nearest neighbour classification

This is implemented in bow_recognition_nearest() function. We also make use of the findnn() function already provided to us to find the nearest point among a set for a given point.

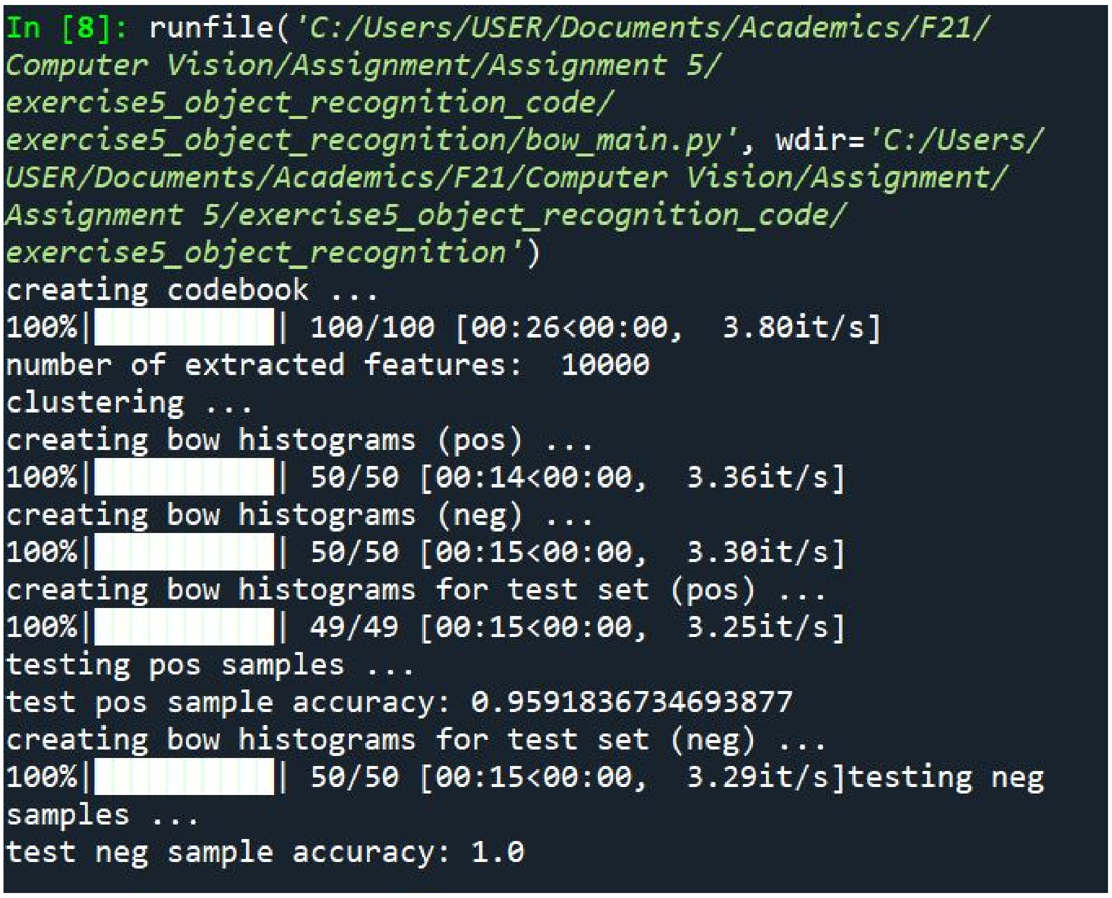
In the above section, we computed the bag of words histogram for each training image. We already have the labels for each training image (whether it contains a car or not). Now, for a new unseen test image, we also calculate its bag of words histogram. Among the training dataset, we find that image which has the closest bag of words histogram to the new test image. The test image is assigned the same label as the closest training image. This is done for all the test images and we get the predictions for each of the test images.

1.6 Results and Observations

There are two hyperparameters which can be adjusted. The first is the number of cluster centers in k-means clustering. If the chosen value of k is too low, then it might lead to underfitting while a high value of k may lead of overfitting. The optimal value we found was equal to $k = 16$.

The second hyperparameter one can play around with is the maximum number of iterations before stopping for the k-means algorithm. We observed that the optimal value of this parameter was $n = 10$, i.e. we run 10 iterations of kmeans before stopping. We get the same result if we run the k-means algorithm multiple times (because the initial values are selected by scikit deterministically for the constant data, as explained in previous sections).

The results we obtained are given in the below screenshot.



```
In [8]: runfile('C:/Users/USER/Documents/Academics/F21/
Computer Vision/Assignment/Assignment 5/
exercise5_object_recognition_code/
exercise5_object_recognition/bow_main.py', wdir='C:/Users/
USER/Documents/Academics/F21/Computer Vision/Assignment/
Assignment 5/exercise5_object_recognition_code/
exercise5_object_recognition')
creating codebook ...
100%|██████████| 100/100 [00:26<00:00,  3.80it/s]
number of extracted features:  10000
clustering ...
creating bow histograms (pos) ...
100%|██████████| 50/50 [00:14<00:00,  3.36it/s]
creating bow histograms (neg) ...
100%|██████████| 50/50 [00:15<00:00,  3.30it/s]
creating bow histograms for test set (pos) ...
100%|██████████| 49/49 [00:15<00:00,  3.25it/s]
testing pos samples ...
test pos sample accuracy: 0.9591836734693877
creating bow histograms for test set (neg) ...
100%|██████████| 50/50 [00:15<00:00,  3.29it/s]testing neg
samples ...
test neg sample accuracy: 1.0
```

Figure 1: Results for Bag of words classification

2 CNN based classification

We were given the skeleton code for training and testing, and we had to fill in the code to construct the given neural network. We mention our implementation briefly along with calculated parameter values for each layer in the neural network. We look at the obtained training and testing results and discuss our observations and conclusions.

2.1 Parameter values for each layer in Neural Network

Our first task is to determine the parameters for the layers in the neural network, e.g. padding and stride sizes for each layer given the input and output dimensions.
In both the cases of convolutional layer and maxpooling, the output dimensions are given by:

$$W_o = \lfloor \frac{W_i + 2P - K}{s} + 1 \rfloor$$
$$H_o = \lfloor \frac{H_i + 2P - K}{s} + 1 \rfloor$$

Here W_i, H_i are width and height for input image, W_o, H_o are width and height for the output image, K is the kernel size, P is the padding and s is the stride. Note that this is valid for both Conv2D and MaxPool2D layers. We are given a network consisting of multiple convolutional blocks. Each of the convolutional block (except for the last one) consists of a Conv2D followed by a ReLU folllowed by a MaxPool2D operation. We are given the sizes of input and output before and after each convolutional block. This allows us to determine the parameters for the Conv2D and MaxPool2D operations using the above formula. The parameters for each of the block were:

1. Convolutional Block 1:
   Conv2D

Parameter	Input channels	Output Channels	Kernel size	Stride	Padding
Value	3	64	3	1	1

   MaxPool2D

Parameter	Kernel size	Stride	Padding
Value	2	2	0

2. Convolutional Block 2:
   Conv2D

Parameter	Input channels	Output Channels	Kernel size	Stride	Padding
Value	64	128	3	1	1

   MaxPool2D

Parameter	Kernel size	Stride	Padding
Value	2	2	0

3. Convolutional Block 3:
   Conv2D

Parameter	Input channels	Output Channels	Kernel size	Stride	Padding
Value	128	256	3	1	1

   MaxPool2D

Parameter	Kernel size	Stride	Padding
Value	2	2	0

4. Convolutional Block 4:
   Conv2D

Parameter	Input channels	Output Channels	Kernel size	Stride	Padding
Value	256	512	3	1	1

   MaxPool2D

Parameter	Kernel size	Stride	Padding
Value	2	2	0

5. Convolutional Block 5:
   Conv2D

Parameter	Input channels	Output Channels	Kernel size	Stride	Padding
Value	512	512	3	1	1

   MaxPool2D

Parameter	Kernel size	Stride	Padding
Value	2	2	0

For the final layer, we have to implement a Linear layer followed by a ReLU, then a dropout followed by another linear layer. We were given freedom to choose the dropout probability as well as the output size for the intermediate linear layer. We chose the dropout probability to be 0.75 and the intermediate linear layer to have an output size of 128. One important thing to note in the implementation is that we have to convert the output of convolutional block 5 to a 2D tensor by squeezing the last 2 axes (which have dimension 1) to feed to the linear layer.

2.2 Hyperparameters and Training

The hyperparameters selected were as follows:

- Batch size: 128
- Number of training epochs: 50
- Learning rate: 0.0001

2.3 Plots

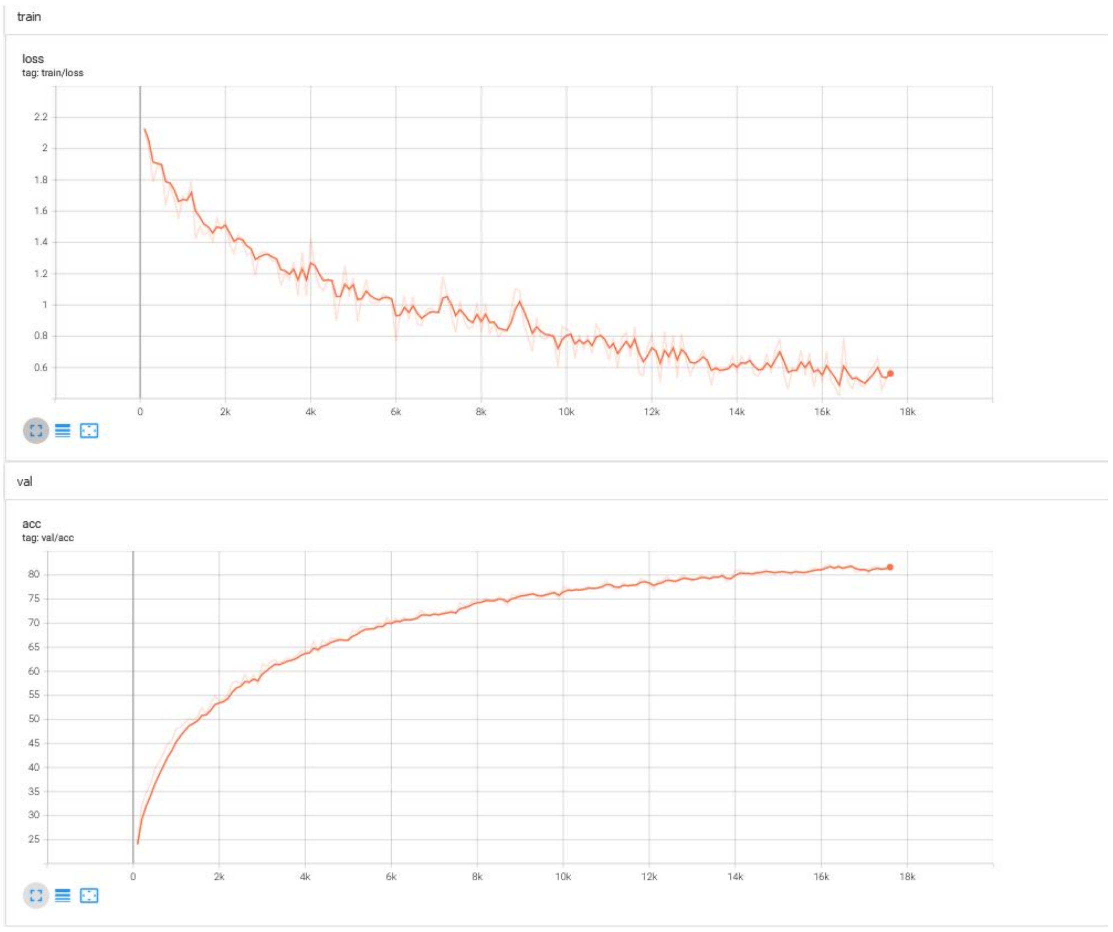


Figure 2: Tensorboard Plots

2.4 Results

Loss after 50 epochs: 0.6024
Validation accuracy: 82.14
Testing accuracy: 81.32