# Computer Vision Lab Assignment 3 Report

lijiaj@student.ethz.ch

October 2023

In this assignment, I was required to complete two tasks: (1) A bag-of-words image classifier and (2) A CNN-based image classifier. I will organize the report by following the same structure.

# 1 Bag-of-words Classifier

My bag-of-words classifier consists of four components/steps: local feature extraction, codebook construction, bag-of-words vector encoding, and nearest neighbour classification.

## 1.1 Local Feature Extraction

Local feature extraction contains two subtasks: feature detection and feature description.

The particular feature detection method I implemented was points on a grid, which conceptually creates a grid on the image with border constraints and extracted grid points as the key features. Specifically, in the `grid_points` function, I first utilized parameters `nPointsX`, `nPointsY`, and `border` to calculate the even spacing between grid points along the $x$ and $y$ direction given the border constraints, respectively. Then a two-layer nested loop was used to compute the exact pixel locations of every grid point in a row-major order. Since in the following code we set the values of `nPointsX` and `nPointsY` both to 10, we had 100 grid points on every image.

The feature description technique utilized to describe the above key grid points was histogram of oriented gradients (HOG), which computes a 128-dimensional vector for every feature point. In the `descriptors_hog` function, the neighbouring 4x4 cells of every grid point were considered and an 8-bin histogram of gradient orientations was calculated for every cell, which led to $4 \times 4 \times 8 = 128$-dimensional vector for every feature point. The size of each cell was determined by the function parameters `cellWidth` and `cellHeight`, which both were defined as 4 in the following code. When filling the missing parts in the given `descriptors_hog` function skeleton, `np.arctan2` was used to compute the angles of the gradients and the `np.histogram` was called for constructing the histogram of gradient orientations for every cell.

## 1.2 Codebook construction

To find the appearance variability of an object class, I constructed a appearance codebook or visual dictionary. First, I extracted local feature descriptors from all training images and then used K-Means algorithm to cluster these feature descriptors. Each

cluster centroid was then considered as a "visual word", a characteristic visual element across different images. In terms of implementation of the `create_codebook` function, I mainly just called previously defined `grid_points` and `descriptors_hog` to obtain feature descriptors of the training images because the other code was already given. The number of clusters and the maximum number of iterations of the K-Means algorithm were introduced as hyperparameters here.

## 1.3 Bag-of-words Vector Encoding

With the codebook constructed in the previous step, every image can be represented as a histogram of visual words.

First, I implemented the `bow_histogram` function that constructs a histogram of visual words for an image given the appearance codebook and the feature vectors of that image. Specifically, every feature vector of the image was assigned to its nearest visual word in the codebook and a histogram was built by counting the number of occurrences of each visual word. In terms of implementation details, I called given `findnn` function to get the indices of the nearest centroids for each feature and used the indices to create a histogram.

Then, I implemented the `create_bow_histograms` function that creates a histogram of visual words for the entire training dataset. As for the implementation details, I basically just called the previously defined `grid_points`, `descriptors_hog`, and `bow_histogram` functions to extract feature vectors and construct the corresponding histogram of visual words for an image in a loop as the other code was already written in the skeleton.

## 1.4 Nearest Neighbor Classification

Finally, I built a bag-of-words nearest neighbour image classifier. A test image was assigned to the category of the training example with nearest histogram of visual words, given the histogram of visual words of the test image and the histograms of visual words of all training images. In the `bow_recognition_nearest` function, I first invoked the provided `findnn` function to calculate the minimum distances between the test image and the positive training samples, as well as between the test image and the negative training samples. Then, the test image was classified as positive if the distance to the positive training samples was smaller. Otherwise, it was considered negative.

## 1.5 Results and Discussion

The bag-of-words classifier described above has two hyperparameters: `k` (the number of clusters in the K-Means algorithm) and `numiter` (the maximum number of iterations of the K-Means algorithm). Different hyperparameter values could lead to different performance. Therefore, I tried two different k values: 10 and 30. For each `k` value, I further tried three different `numiter` values starting at 5 and going up to 15 in intervals of 5. The performance of the bag-of-words nearest neighbour image classifier under different hyperparameter values is summarized in the Table 1 and 2 (the percentage accuracy is rounded to the integer precision). During these experiments, the random seed was fixed to 42 for reproducibility. All the screenshots of logging of these experiments are listed in the Appendix section.

| k = 10 | test positive sample accuracy | test negative sample accuracy |
|---|---|---|
| numiter = 5 | 96% | 98% |
| numiter = 10 | 94% | 100% |
| numiter = 15 | 96% | 100% |

Table 1: Test accuracy with k = 10

| k = 30 | test positive sample accuracy | test negative sample accuracy |
|---|---|---|
| numiter = 5 | 96% | 96% |
| numiter = 10 | 94% | 96% |
| numiter = 15 | 90% | 90% |

Table 2: Test accuracy with k = 30

We can draw some key observations from these tables. First, regardless of the specific values of `k` and `numiter`, in most cases the accuracy of negative class test samples is higher than that of positive class. Second, higher `numiter` values do not always lead to a higher overall accuracy given a fixed `k` value. Third, when the value of `numiter` is fixed, the test accuracy does not seem to improve significantly as the value of `k` increases. Therefore, I set `k` to 10 and `numiter` to 15 in the submitted code for best accuracy.

# 2 CNN-based Classifier

## 2.1 A Simplified Version of VGG Network

I implemented a simplified version of VGG neural network in the `vgg_simplified.py` file as required. This simplified VGG network consisted of 5 convolutional blocks and a classifier block. Every convolutional block was composed of a convolutoinal layer, a ReLU layer, and a max-pooling layer. The classifier block contained a linear layer, ReLU layer, a dropout layer, and a linear layer. In the `__init__` method of the VGG class, I created every aforementioned block as a `nn.Sequential` object. Then I passed the input image batch through these blocks in order in the `forward` method of the VGG class to obtain the model output.

## 2.2 Training and Testing

Since the training script was already given, I didn't modify any parameter in the training script as I believed these default values could give me a good enough result. The training loss curve is given in Figure 1 and we can see that the training loss is generally decreasing during the whole training process in spite of certain slight fluctuation. The validation accuracy curve is presented in Figure 2 showing the validation accuracy steadily increased over 80%.

In terms of testing, the situation was similar. I only adjusted the `model_path` parameter in the testing script to the location of best model found during training as other parts of the code were already implemented by the teaching team. The final test accuracy was 82.31% as in Figure 3.


Figure 1: Training Loss Curve

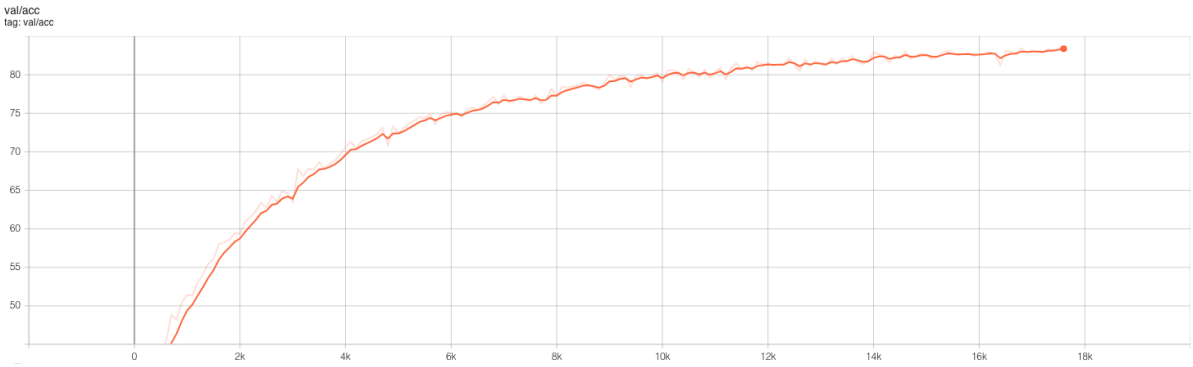


Figure 2: Validation Accuracy Curve

```
(ex4_cuda) root@autodl-container-beda118efa-7324c7b0:~/exercise4_object_recognition_code# python3 test_cifar10_vgg.py
[INFO] test set loaded, 10000 samples in total.
79it [00:03, 21.60it/s]
test accuracy: 82.31
```

Figure 3: Test Accuracy

# 3 Appendix

The screenshots of logging of `bow_main.py` file under different hyperparameter values are listed below for your reference.

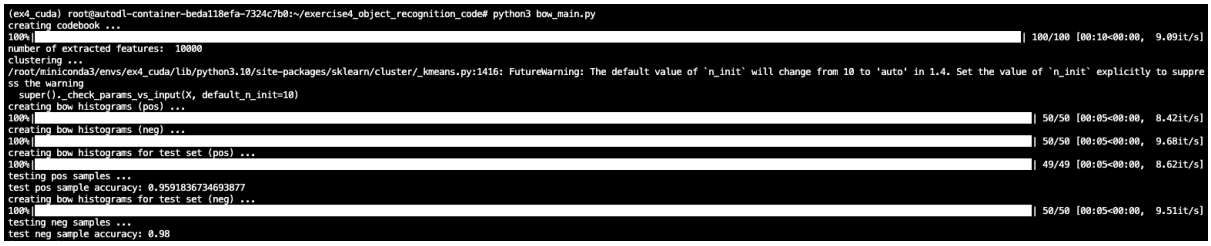


Figure 4: Logging with k = 10 numiter = 5


Figure 5: Logging with k = 10 numiter = 10

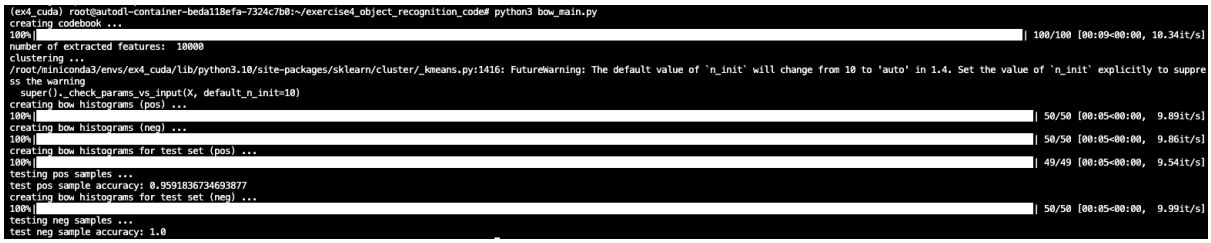


Figure 6: Logging with k = 10 numiter = 15

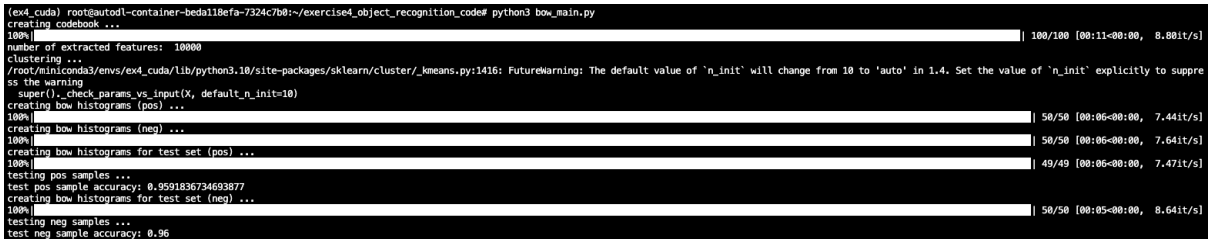


Figure 7: Logging with k = 30 numiter = 5

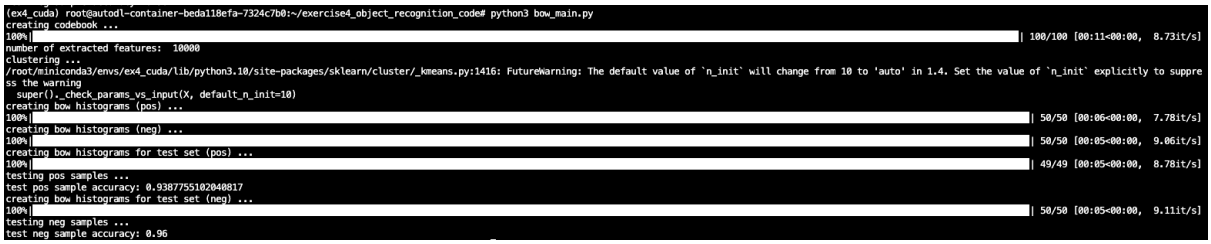


Figure 8: Logging with k = 30 numiter = 10

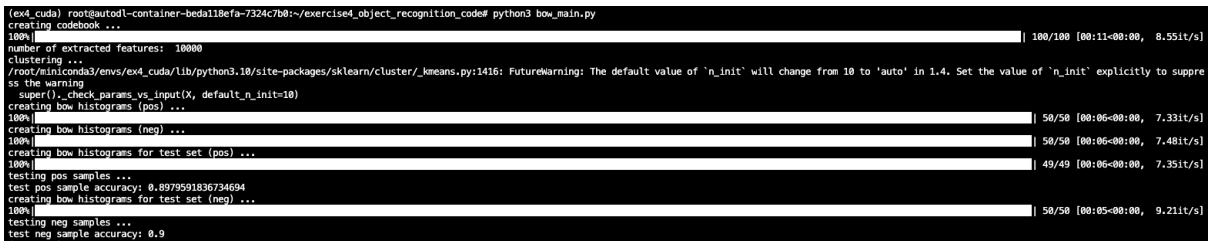


Figure 9: Logging with k = 30 numiter = 15