Exercise 4: Object Recognition

Bag-of-Words and CNN-Based Classifiers

Orion Junkins

263-5902-00L Computer Vision

Marc Pollefeys, Siyu Tang & Fisher Yu

November 10, 2023

1. Introduction

This lab approaches image classification in two ways. First, it applies a traditional bag-of-words approach to a relatively easy binary classification problem. Second, it applies a simplified version of the VGG network to the CIFAR-10 dataset, a challenging multiclass classification problem.

2. Bag-of-Words Classifier

A Bag-of-Words is designed to identify if an image does or does not show the rear profile of a car.

2.1. Local Feature Extraction

2.1.1 Feature detection

Given an image, a border size, and a desired number of points in both the x and y direction, impose a grid upon the image and return the coordinates of the points. First, identify the grid bounds as *0 + border* and *max - border* in the X and Y directions. Find the difference between the two bounds and divide by the corresponding number of points to get grid cell size. Iterate from the lower bounds to the upper bounds with a step size of the grid cell size along both axes to generate a list of coordinates.

Note that this approach generally assumes that the border and desired number of points are chosen s.t. *image max dim - 2 * border % number of points = 0* along both axes. If this condition is not met, some amount of the image will be ignored.

2.1.2 Feature Description

Next, generate a Histogram of Oriented Gradients (HOG) descriptor for a given image and a grid of N points chosen as described above. For the entire image, calculate gradients along both axes I_x and I_y, using Sobel filters. For each grid point, create a 128-value vector descriptor of the local region using I_x and I_y.

For every grid point, isolate a 4x4 grid of cells centered there with cells of the specified width and height (both set to 4 pixels by default). For all individual pixels within each cell (16 total in this case), find the angle of the gradient vector Θ at that point:

$$\theta(x, y) = arctan[I_x(x, y), I_y(x, y)]$$

Discretize these Θ values into an 8-bin histogram. This histogram is an 8-value descriptor for that cell. For every grid point, concatenate the 8-value descriptors for all 16 cells into a single vector. This 128-value vector is the HOG descriptor for the current grid point.

Find this descriptor for all N grid points and return the Nx128 matrix containing all HOG descriptors for the given image.

2.2. Codebook construction

Given a collection of images, construct an *appearance codebook*. For every image within the given directory, say D images, find the N x 128 matrix of HOG descriptors using the procedure described above. Combine all of the descriptors into a 2D array of shape (N*D, 128) and apply a K-means clustering algorithm with the specified "*k*" and "*numiter*" parameters. These clusters are the *visual words* for our bag of words approach.

This codebook is created with all training samples, both positively classified and negatively classified.

2.3. Bag-of-words Vector Encoding

2.3.1 Bag-of-Words histogram

Using this codebook, and the extracted features for a particular image, a new representation of size k can be found for any image. For all extracted features, find the closest corresponding entry (equivalently, the closest *visual word* or *cluster center*) in the codebook. Find the number of features corresponding to each of the k entries and record a histogram of these values.

Conceptually, this histogram signifies the number of occurrences of various visual patches (defined by the HOGs within each patch) in the provided image.

2.3.2 Processing a directory with training examples

To train a binary classifier, the user must provide a directory of training samples for each class. Rather than combining all images, as is done for codebook creation, images within each class are processed separately.

For every image within a particular class, the Bag-of-words histogram of size k is found. These histograms are stored in a 2D matrix of dimension k x D, where D is the size of the training set in the current class.

This is performed for both positive and negative training sets, yielding 2 k x D matrices.

2.4 Nearest Neighbor Classification

Now, classification can be performed. Two directories of testing images are provided - one containing positive samples and one containing negative samples. Each of these directories is processed using the same function used in training to produce a k x D matrix that represents all histograms of size k for the D images in the directory. For each of these D histograms, the most similar training sample is identified as the closest histogram in the codebook. Each sample is labeled according to the label of its nearest neighbor in the codebook.

For all positive and negative samples, the number of correct classifications is found, and accuracy is calculated.

2.5 Hyper-Parameter Tuning

Approach

Two parameters must be tuned: *k* (the number of cluster centers, the size of the codebook vocabulary) and *numiter* (the maximum number of iterations used in the K-Means clustering).

While some heuristics are useful for choosing these parameters, I opted for a grid search given the relatively short training times.

All combinations of the following values were evaluated:

Parameter	Values
Number of Clusters (*k*)	1, 2, 4, 8, 16, 32, 64, 128
Number of iterations (*numiter*)	4, 8, 16, 32,

Results

The appendix (Section 4.1) contains raw grid search results data.

Achieving high negative accuracy is a relatively trivial task, with many parameter combinations yielding 100% accuracy. Combinations that do not yield 100% negative accuracy are excluded. The remaining parameter combinations are evaluated by their positive accuracy. The chart on the following page (Figure 1) shows the positive accuracy results across various parameter combinations.

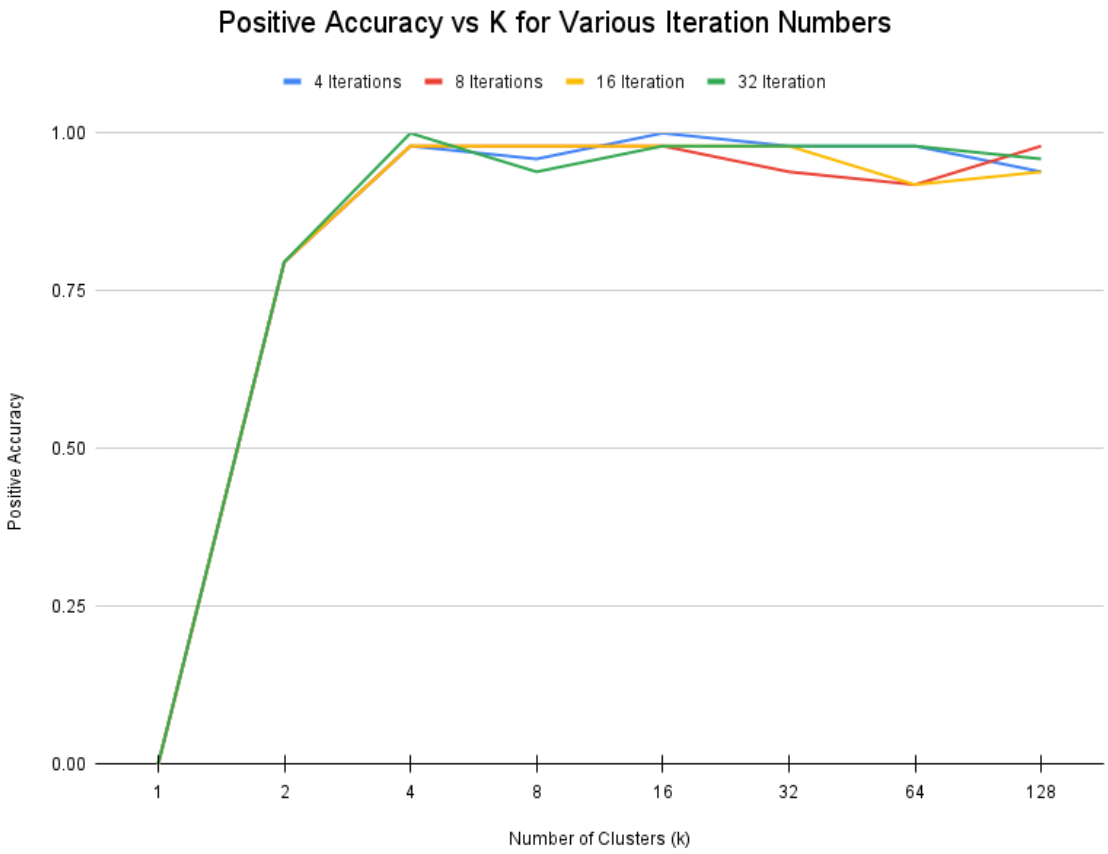

Figure 1: Grid Search results showing the relationship between k and positive accuracy across various iteration counts

Selected Values

Number of iterations: The impact of *numiter* on final accuracy seems negligible, given that it is above some threshold. There are subtle differences in the final score, but no value is clearly superior. Given the rapid computation time, the lack of computational constraints, and the fact that the underlying implementation will halt at convergence regardless, I select 16 as the final value. Experimental results show that this is more than sufficient. This is chosen knowing that convergence will generally occur well before this, and lower values could likely be used without hurting accuracy if there was a reason to do so.

Number of Clusters: Increasing clusters from 0 to 4 increases accuracy substantially, but returns are diminishing beyond this. No improvement occurs from 4 to 16. Above 16, accuracy begins to decrease. Thus, I select 4, as the minimum value needed for high accuracy.

Final parameters: *numiter* = 16, *k* = 4

2.6 Final Ouput

Figure 2 below shows the output of running *bow_main.py* with the chosen parameters.

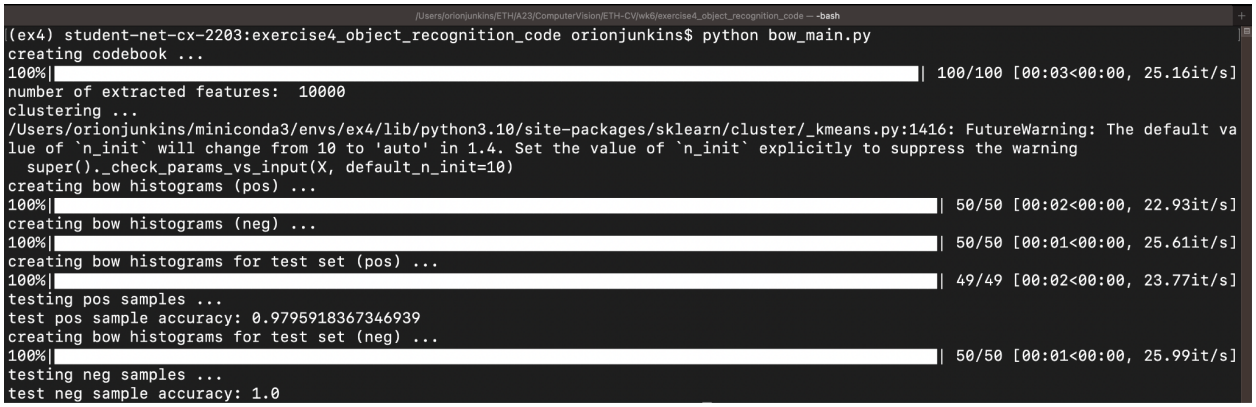

Figure 2: Final output

3. CNN-based Classifier

3.1 A Simplified version of VGG Network

Next, this lab implements a simplified version of the VGG Network and applies it to classification of the CIFAR-10 dataset.

The network contains five convolutional layers, each consisting of a 2D Convolutional layer, a ReLU activation function, and a 2D max pooling. The max pooling causes the output dimensions to be cut in half at each layer, resulting in a ‘funnel’ shape for the network. To preserve the signal while the dimensionality is reduced, the first four convolution layers double the number of channels.

The final classification uses the sequential combination of a 512-neuron linear layer, ReLU, Dropout, and a final 10-neuron linear layer.

3.2 Training and Testing

3.2.1 Training

The model trained for 80 epochs (~28,000 steps) in 1.443 hours on a Mac M2 with the MPS GPU backend.

Training loss continues to drop up to 80 epochs. However, validation loss levels off much earlier, indicating overfitting. Tensorboard outputs are visible in Figures 3 and 4 on the following page.

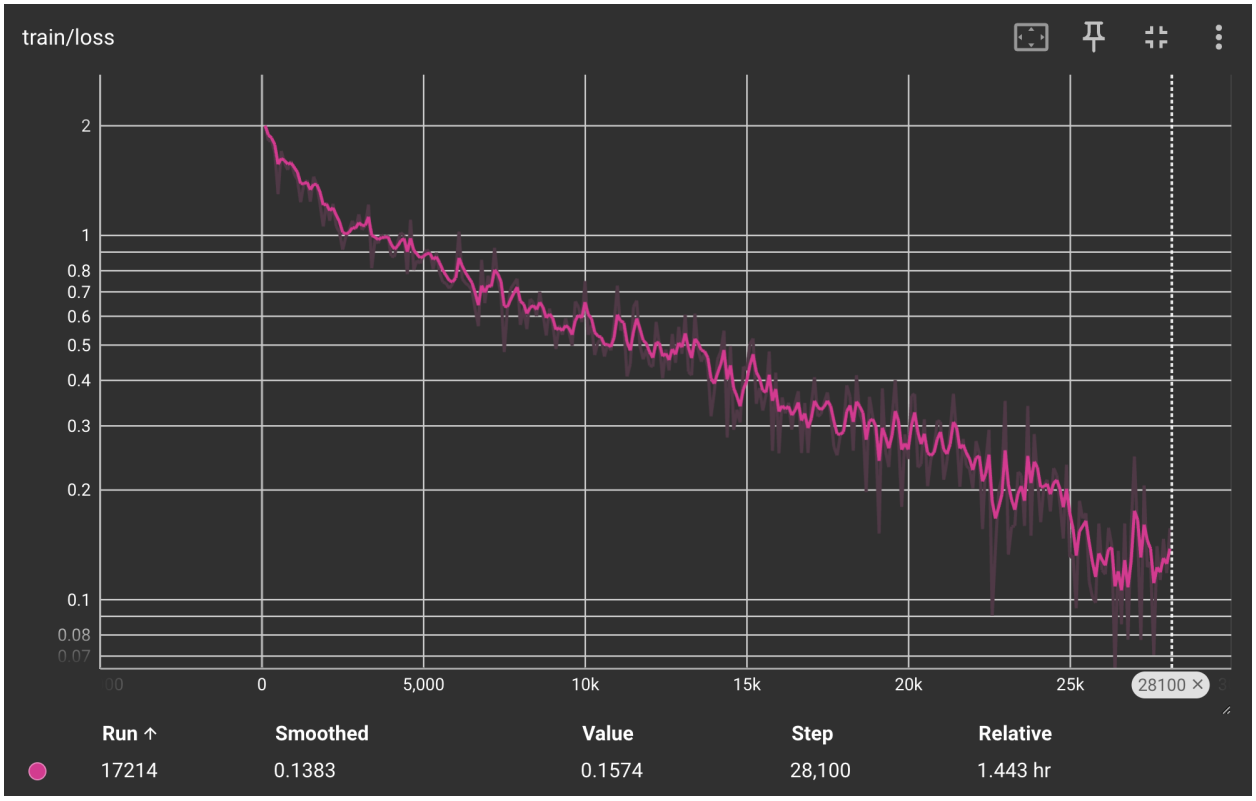

Figure 3: Tensorboard output for training loss

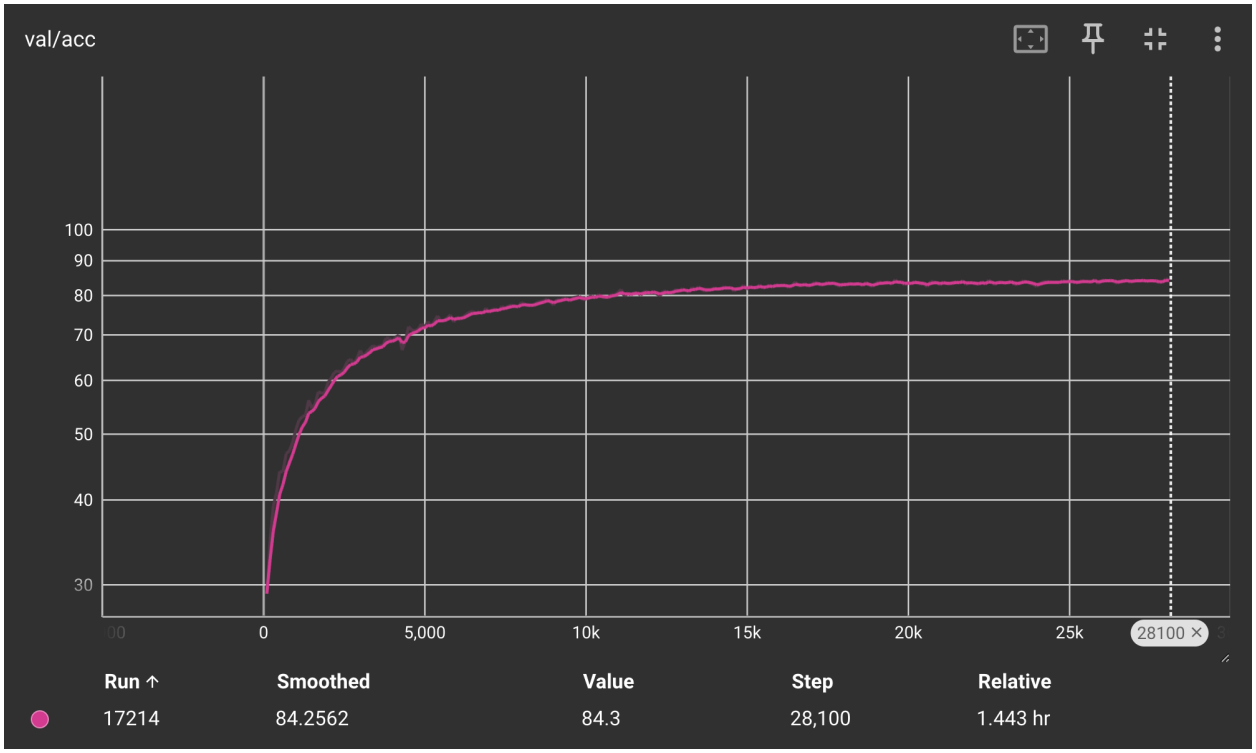

Figure 4: Tensorboard output for validation loss

3.2.1 Testing

To ensure reproducibility, test inference is run on the CPU. Figure 5 below shows the output of running *test_cifar10_vgg.py* with the chosen parameters.

```
(ex4) student-net-cx-2203:exercise4_object_recognition_code orionjunkins$ python test_cifar10_vgg.py
[INFO] test set loaded, 10000 samples in total.
79it [00:07, 10.34it/s]
test accuracy: 83.09
```

Figure 5: Final output

4. Appendix

4.1 Full Results For BoW Grid Search

Number of Clusters (*k*)	Number of iterations (*numiter*)	Positive Accuracy	Negative Accuracy
1	4	0	1
1	8	0	1
1	16	0	1
1	32	0	1
2	4	0.7959183673	1
2	8	0.7959183673	1
2	16	0.7959183673	1
2	32	0.7959183673	1
4	4	0.9795918367	1
4	8	0.9795918367	1
4	16	0.9795918367	1
4	32	1	1
8	4	0.9591836735	1
8	8	0.9795918367	0.98
8	16	0.9795918367	1
8	32	0.9387755102	1
16	4	1	1
16	8	0.9795918367	0.98
16	16	0.9795918367	1
16	32	0.9795918367	1
32	4	0.9795918367	1
32	8	0.9387755102	0.96
32	16	0.9795918367	1
32	32	0.9795918367	1
64	4	0.9795918367	0.98
64	8	0.9183673469	1
64	16	0.9183673469	1
64	32	0.9795918367	0.92
128	4	0.9387755102	1

128	8	0.9795918367	1
128	16	0.9387755102	1
128	32	0.9591836735	1