

# CV Lab 03: Object Classification

James Liu

November 2023

## 1 Introduction

The aim of this lab was to gain familiarity with both classical Bag-of-Words (BoW) and Deep Learning (DL) techniques to perform image classification.

## 2 Bag-of-Words

The implementation of this BoW classifier is found in `bow_main.py`.

### 2.1 Local Feature Extraction

#### 2.1.1 Feature detection - points on a regular grid

Here, to define locations of feature points, we simply sample points on the image from a regular grid. This was implemented in the `grid_points()` method, with the core logic shown below:

```
def grid_points(img, nPointsX, nPointsY, border):
    ... # truncated
    # Calculate the spacing between grid points in both dimensions
    x_spacing = (width - 2 * border) / (nPointsX - 1)
    y_spacing = (height - 2 * border) / (nPointsY - 1)

    vPoints = []
    # Generate grid points
    for i in range(nPointsY):
        for j in range(nPointsX):
            # Calculate the coordinates of the grid point
            x = int(border + j * x_spacing)
            y = int(border + i * y_spacing)
            vPoints.append((x, y))
    return np.array(vPoints)
```

where we first define the spacing between grid points in the x and y directions, taking into account the border, and number of desired grid points. `vPoints` is then an array of size `[nPointsX*nPointsY, 2]`.

#### 2.1.2 Feature description - histogram of gradients

To describe each feature, we use a histogram of oriented gradients (HOG). This is computed by `descriptors_hog()`.

```
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)

    descriptors = [] # list of descriptors for the current image, each entry is one 128-d vector for
    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 and end x, y indices for a given cell comprised of w*h pixels
        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

        # Extract gradients for image patch
        patch_y = grad_y[start_y:end_y, start_x:end_x]
        patch_x = grad_x[start_y:end_y, start_x:end_x]

        # Orientations for each pixel in cell, from 0 to 360 degrees
        orientations = np.arctan2(patch_y, patch_x) * (180/np.pi) + 180

        # Ensure dominant orientation is at 0 degrees
        dominant_orientation = np.mean(orientations)
        orientations_shifted = (orientations - dominant_orientation) % 360

        # Compute histogram of orientations
        hist, bin_edges = np.histogram(orientations_shifted, bins=nBins)
        desc.append(hist)

desc = np.array(desc).flatten()
descriptors.append(desc)

descriptors = np.asarray(descriptors) # [nPointsX*nPointsY, 128], descriptor for the current image
return descriptors

```

For each feature point in `vPoints`, we look at the surrounding 4x4 neighbour of "cells", where each cell is a region of `cellWidth` x `cellHeight` pixels. We then extract the x and y image gradients for each cell, and calculate the gradient orientation for each pixel in this cell or patch via trigonometry,

$$\theta = \arctan \frac{F_y}{F_x} \quad (1)$$

where  $F_y$  and  $F_x$  are the gradients in the y and x direction respectively. Note the extra factors  $*(180/\text{np.pi})+180$  to rescale the orientations from 0 to 360 degrees, instead of  $-\pi$  to  $+\pi$  radians. A histogram is then taken over a flattened version of the `orientations` grid, ensuring first that we shift the orientations relative to the dominant direction. This ensures rotational invariance of the descriptors.

The 8-bin histograms are computed for each of the 4x4=16 cells assigned to the feature point and concatenated, resulting in a 128-d feature vector for each feature point.

## 2.2 Codebook construction

We then use the two previous methods to generate feature points and associated feature descriptors across all training images.

```

for i in tqdm(range(n_imgs)):
    img = cv2.imread(v_img_names[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

    # Get grid points
    v_points = grid_points(img, n_points_x, n_points_y, border)

```

```

# Compute descriptors
descriptors = descriptors_hog(img, vPoints, cellWidth, cellHeight) # [num_descriptors, 128]
vFeatures.append(descriptors)

```

In each iteration of this for loop, we use `cv2.imread()` to load the training image, call `grid_points()`, and then `descriptors_hog()` to generate 128-d descriptors for each feature point. We can then append these descriptors to the `vFeatures` list. This results in a `[n_imgs, n_vPoints, 128]` array, which we can reshape to `[n_imgs*n_vPoints, 128]`, storing all the descriptors across the training images. The `KMeans` class from `sklearn` is then used to cluster the descriptors, forming a codebook with  $k$  "words", where the words are the  $k$  cluster centers in feature space.

## 2.3 Bag of Words encoding

### 2.3.1 Bag-of-Words histogram

Now that we have the codebook, we can generate a histogram over the visual words for a given image.

```

def bow_histogram(vFeatures, vCenters):
    """
    :param vFeatures: MxD matrix containing M feature vectors of dim. D
    :param vCenters: NxD matrix containing N cluster centers of dim. D
    :return: histo: N-dim. numpy vector containing the resulting BoW activation histogram.
    """

    # Get the number of cluster centers (N) and feature vectors (M)
    num_centers = vCenters.shape[0]
    num_features = vFeatures.shape[0]

    # Initialize histogram with zeros
    histo = np.zeros(num_centers)

    # Assign each feature vector to the nearest cluster center
    for i in range(num_features):
        feature_vector = vFeatures[i, :]

        # Calculate euclidean distances
        distances = np.linalg.norm(vCenters - feature_vector, axis=1)

        # Find the index of the nearest cluster center
        nearest_center_index = np.argmin(distances)

        # Increment the corresponding bin in the histogram
        histo[nearest_center_index] += 1

    return histo

```

The function takes in the descriptors of an image, and the feature vectors for the words. Within the `for` loop, the euclidean distance between current descriptor and the word vectors are computed by taking the L2 norm of the difference vectors,  $\|x - y\|_2$ . Note the `axis=1` arg, ensuring the norm is taken over the feature space dimension,  $D$ . `distances` is of size `[N, D]`. The index of the smallest distance is then returned with `np.argmin(distance)`, and the feature is assigned to the corresponding word, incrementing the histogram count.

### 2.3.2 Processing a directory with training examples

Next, we create a BoW histogram for every training image.

```

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

# 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]

    vPoints = grid_points(img, nPointsX, nPointsY, border)
    vFeatures = descriptors_hog(img, vPoints, cellWidth, cellHeight) # [k, 128]
    bow_histo = bow_histogram(vFeatures, vCenters) # [k, ]
    vBoW.append(bow_histo)

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

```

For each image, we again call `grid_points()` and `descriptors_hog()` to extract feature descriptors. `bow_histogram()` is then called to generate the BoW histogram for the image (size `[k,]`). The histogram is then appended to the list `vBoW`, size `[n_imgs, k]`. This stores the number of occurrences of each visual word for each image.

## 2.4 Nearest Neighbour Classification

Finally, to classify a test image, we compute its BoW histogram. Then we assign it to its nearest neighbour in the training BoW histograms.

```

def bow_recognition_nearest(histogram, vBoWPos, vBoWNeg):
    """
    :param histogram: bag-of-words histogram of a test image, [1, k]
    :param vBoWPos: bag-of-words histograms of positive training images, [n_imgs, k]
    :param vBoWNeg: bag-of-words histograms of negative training images, [n_imgs, k]
    :return: sLabel: predicted result of the test image, 0(without car)/1(with car)
    """
    # Find the nearest neighbor in the positive and negative sets and decide based on this neighbor

    DistPos = np.linalg.norm(vBoWPos-histogram, axis=1) # norm([n_imgs, k], axis=1) -> [n_imgs,]
    DistPos = np.min(DistPos) # Take smallest distance

    DistNeg = np.linalg.norm(vBoWNeg-histogram, axis=1)
    DistNeg = np.min(DistNeg)

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

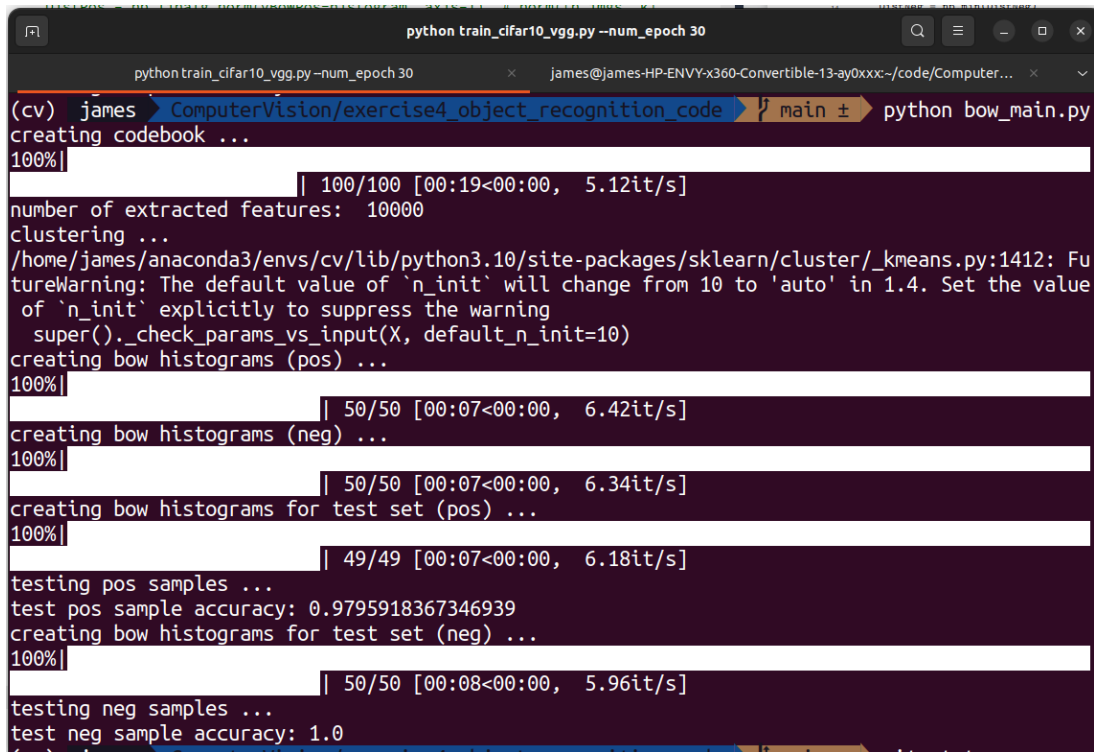
```

Here, we use a similar method of finding the smallest distance between a vector and another set of vectors. Note though `histogram` is `[1, k]` and `vBoWPos` is `[n_imgs, k]`, `vBoWPos-histogram` is cast to `[n_imgs, k]`, where each row of `vBoWPos` is subtracted by `histogram`. Taking the L2-norm over `axis=1` returns a `[n_imgs,]` array.

We compare the minimum distances between the positive and negative training examples, and assign `sLabel=1` if the minimum distance between the test histogram and a positive training example histogram is smaller than that of the negative example, i.e. the nearest neighbour is positive.

## 2.5 Classification Results

Shown in Figure 1 are the logging results after running `$ python bow_main.py`.



```
python train_cifar10_vgg.py --num_epoch 30
python train_cifar10_vgg.py --num_epoch 30
james@james-HP-ENVY-x360-Convertible-13-ay0xxx:~/code/Computer...
(cv) james ComputerVision/exercise4_object_recognition_code main python bow_main.py
creating codebook ...
100%|████████████████████████████████████████████████████████████████████████████████| 100/100 [00:19<00:00, 5.12it/s]
number of extracted features: 10000
clustering ...
/home/james/anaconda3/envs/cv/lib/python3.10/site-packages/sklearn/cluster/_kmeans.py:1412: FutureWarning: The default value of 'n_init' will change from 10 to 'auto' in 1.4. Set the value of 'n_init' explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
creating bow histograms (pos) ...
100%|████████████████████████████████████████████████████████████████████████████████| 50/50 [00:07<00:00, 6.42it/s]
creating bow histograms (neg) ...
100%|████████████████████████████████████████████████████████████████████████████████| 50/50 [00:07<00:00, 6.34it/s]
creating bow histograms for test set (pos) ...
100%|████████████████████████████████████████████████████████████████████████████████| 49/49 [00:07<00:00, 6.18it/s]
testing pos samples ...
test pos sample accuracy: 0.9795918367346939
creating bow histograms for test set (neg) ...
100%|████████████████████████████████████████████████████████████████████████████████| 50/50 [00:08<00:00, 5.96it/s]
testing neg samples ...
test neg sample accuracy: 1.0
```

Figure 1: Logging results.

Using `k=10` words/clusters and `iters=100`, we achieved 98.0% and 100% test accuracy on positive and negative examples respectively. This suggests that 10 words was enough to successfully classify the car images.

## 3 CNN-based classifier

We now focus on the use of DL as opposed to traditional BoW techniques for image classification.

### 3.1 Simplified version of VGG Network

The following was used to implement a simplified version of the VGG Network.

```
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

        # input shape: [bs, 3, 32, 32]
        # output shape: [bs, 10]

        self.conv_block1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
```

```

self.conv_block2 = nn.Sequential(
    nn.Conv2d(64, 128, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2)
)
self.conv_block3 = nn.Sequential(
    nn.Conv2d(128, 256, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2)
)
self.conv_block4 = nn.Sequential(
    nn.Conv2d(256, 512, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2)
)
self.conv_block5 = nn.Sequential(
    nn.Conv2d(512, 512, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2)
)
self.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(512, fc_layer),
    nn.ReLU(),
    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]
    """
    score = None
    # todo
    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)
    score = self.classifier(x)

    return score

```

In the `__init__()` method, we instantiate a series of `nn.Sequential()` objects in the class attributes, in order to encapsulate the convolutional layers into units. Notice how the number of input and output channels chain from layer to layer, e.g. the first `nn.Conv2d()` layer takes in 3 input channels and outputs 64 channels, resulting in the next `nn.Conv2d()` taking in 64 input channels and so on. Also note the `nn.Flatten()` layer to ensure 1-D inputs into the linear layers.

The `forward()` method then simply repeatedly passes the input `x` through each `conv_block`.

## 3.2 Training

To train, we simply run the training script given, adjusting the length of training.

```
$ python train_cifar10_vgg.py --num-epoch 30
```

producing `params.json`:

```
{
  "batch_size": 128,
  "fc_layer": 512,
  "log_step": 100,
  "lr": 0.0001,
  "num_epoch": 30,
  "root": "data/data_cnn/cifar-10-batches-py",
  "save_dir": "runs",
  "val_step": 100
}
```

The learning curves in Figures 2 and 3 were obtained after approximately 8 epochs, taking  $\approx 4$  hours.

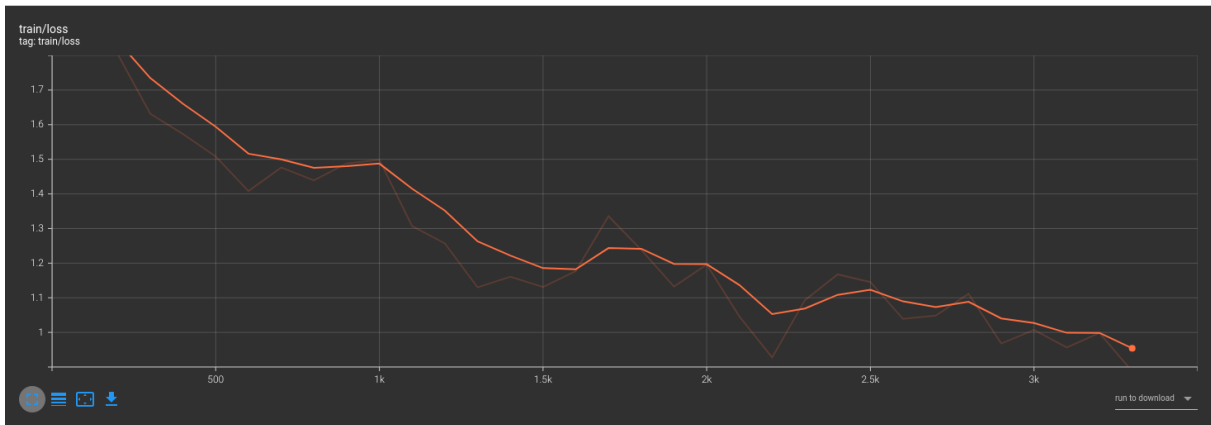


Figure 2: Training loss against training iterations.

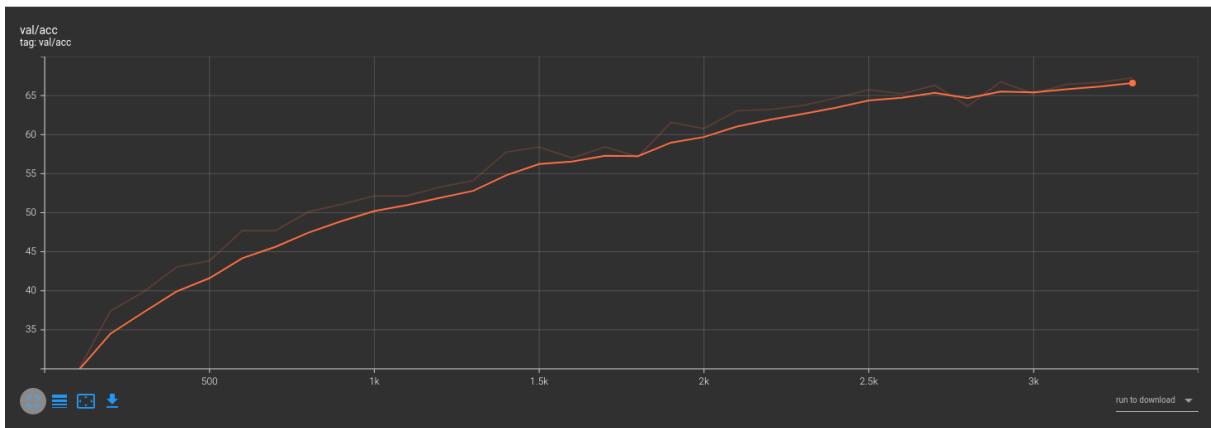


Figure 3: Validation set accuracy against training iterations.

Note: 50000 training images with `batch_size=128` is approximately 390 minibatches per epoch. A **training iteration** refers to one gradient update over a minibatch.  $\Rightarrow 3300 \text{ steps} \div 390 \approx 8 \text{ epochs}$ .

### 3.3 Testing

We then run the command `python test_cifar10_vgg.py` to run the test script on our saved model.

The test set accuracy achieved was 66.03%.