assn2_part1

October 8, 2021

1 Assignment 2

2 Part 1: Image segmentation

2.1 Question 1: Color clustering with K-means (15 points)

Image segmentation is the process of partitioning an image into various regions of pixels that are more meaningful and easier to understand/analyze. It is often used to specify boundaries and separate the most important object(s) from the rest. One way to segment images is to use K-means clustering to cluster image regions with similar colors. Remember that K-means clustering is an unsupervised algorithm that partitions the given data into K clusters based on some definition of similarity.

Do this:

1-a. Implement the K-means clustering algorithm in `kmeans.py`. (10 points)

1-b. Experiment with different values of K. Discuss which value of K seems to be the best for each of the test images. (5 points)

[1]:
```
import cv2, pickle
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
# Here are images you will use to check your implementation.
filenames = ['cat.jpeg', 'dog.jpeg',  'nature.jpeg', 'nyc.jpeg', 'princeton.
 ↪jpeg']

plt.figure(figsize=(4*len(filenames), 5))
for i in range(len(filenames)):
    img = cv2.imread(filenames[i])
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    height, width, nchannel = img.shape
    scale = 500/max(height, width) # The longer side will be resized to 500
    img = cv2.resize(img, (int(width*scale), int(height*scale)))
    plt.subplot(1, len(filenames), i+1)
    plt.imshow(img); plt.axis('off')
plt.show()
```





2.1.1 1-a. Implement the K-means clustering algorithm in `kmeans.py` (10 points)

After implementing the K-means clustering algorithm in `kmeans.py`, use the below code to segment a given image into regions with similar colors.

[3]:
```
# Import your implementation
from kmeans import kmeans
```

[4]:
```
def run_kmeans(K, niter, filename):

    # Load and transform an image
    img = cv2.imread(filename)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    height, width, nchannel = img.shape
    scale = 500/max(height, width) # The longer side will be resized to 500
    img = cv2.resize(img, (int(width*scale), int(height*scale)))

    # Vectorize the image
    x = img.reshape((-1, 3)).astype(np.float32)

    # Run K-means clustering
    labels, centroids = kmeans(x, K, niter)

    # Produce the resulting image segmentation.
    centroids = np.uint8(centroids)
    labels = np.int8(labels)
    result = centroids[labels.flatten()]
    result_image = result.reshape((img.shape))

    # Visualize the original image and the segmentation.
    plt.figure(figsize=(10, 10))
    plt.subplot(1,2,1)
    plt.imshow(img); plt.title('Original Image'); plt.axis('off')
    plt.subplot(1,2,2)
    plt.imshow(result_image); plt.title('Segmented Image (K={})'.format(K));␣
↪plt.axis('off')
```

```
    plt.show()
```

[9]:
```
# Define the arguments
K = 5
niter = 20
filename = 'example.jpeg'

# Run K-means
run_kmeans(K, niter, filename)
```

2.1.2 b) Experiment with different values of K. Discuss which value of K seems to be the best for each of the test images. (5 points)

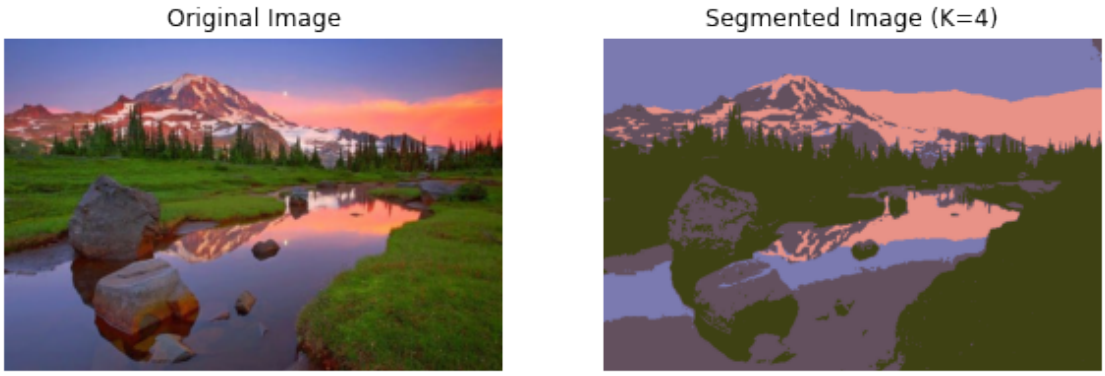
[18]:
```
# Define the arguments
K = 4
niter = 20

# Run K-means
for i in range(len(filenames)):
    run_kmeans(K, niter, filenames[i])
```

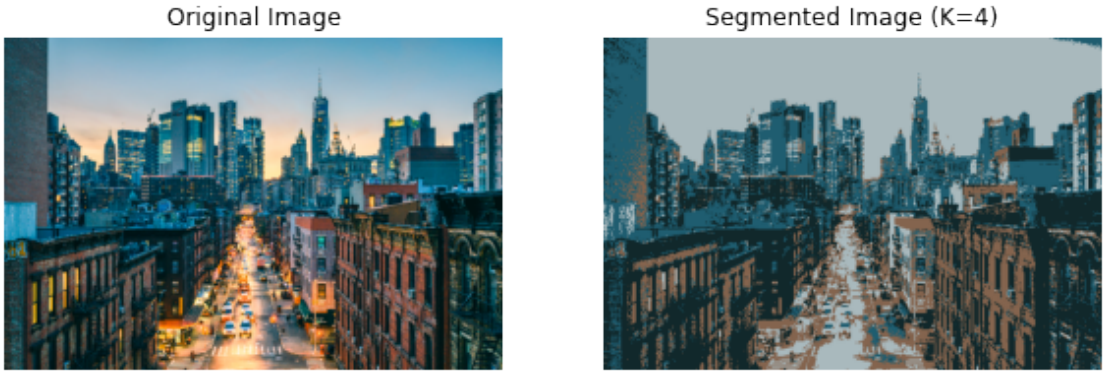
Original Image

Segmented Image (K=4)

Original Image

Segmented Image (K=4)





[33]:
```
# Define the arguments
K = 4
niter = 20

# Run K-means
```

```
for i in range(2, 11):
    run_kmeans(i, niter, 'princeton.jpeg')
```

Original Image

Segmented Image (K=2)

Original Image

Segmented Image (K=3)

Original Image

Segmented Image (K=4)

Original Image

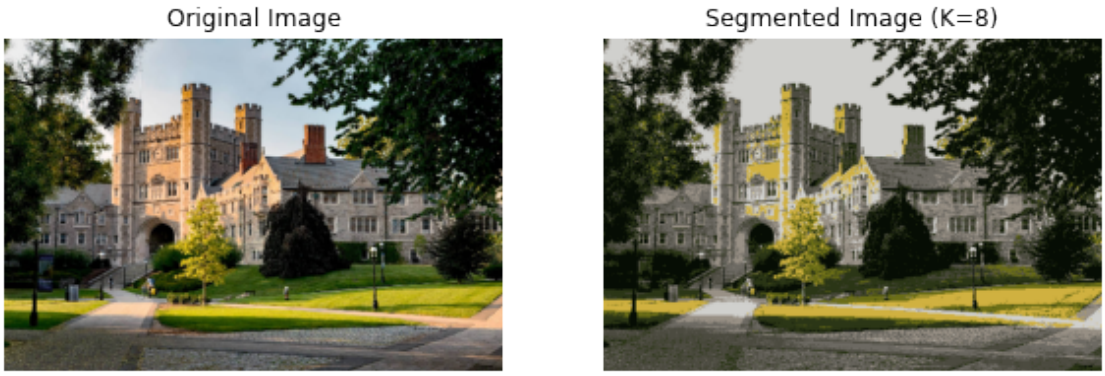
Segmented Image (K=5)

Original Image

Segmented Image (K=6)

Original Image

Segmented Image (K=7)



2.1.3 Answers to 1b

For the cat, k = 3 works best because starting from k = 4, the black region up top starts to get segmented into two with one of the two regions being lighter which isn't a good representation of the region at the top. Moreover, k = 3 is enough to capture the details and difference in color on the cat and it represents the ground well enough. For the dog, k = 6 works best because it correctly

captures the different gradients in the grass and the black region at the top. It also takes into account the different colors of the dog. For the nature image, k = 5 works best because it segments the sky properly and segments the forest and grass immediately in front properly. It also has the outline of the rocks and mountain. k = 4 has the grass and forest blended and k = 6 and up starts to segment the sky too much s.t. it doesn't work For NYC, k = 4 works best because starting from k = 5, we start to get brighter yellow tones which causes the sky to be segmented into two regions but it represents the gradient in the original sky badly. Moreover, even in k = 4, there is already a muted yellow which is sufficient to segment the street lights and lighted building walls, and the city is segmented well from the street below. For Princeton, k = 5 works best because we start getting the green tones. So, the one tree in the middle is no longer muted and looks very similar in color to the original image. We also get the lighted region of the building, a good segmentation of the other greenery and the sky and building, and finally the ground shadows and lightings are captured.

assn2_part2

October 8, 2021

1 Assignment 2

2 Part 2: Classic recognition

Questions 2–7 are coding questions. You will implement different image representations and train linear classifiers with them. You will start with more flexible image representations and progressively move onto more rigid representations. Questions 8-10 are written questions to be answered in the PDF. You will report and reflect on the results, analyze the pros/cons of each representation and discuss possible improvements.

[1]:
```
import os, pickle
import numpy as np
import cv2
import matplotlib.pyplot as plt
import time

from utils import get_CIFAR10_data, train
```

2.1 Setup

We provide you with a linear (softmax) classifier, as well as code to load the CIFAR-10 dataset. The CIFAR-10 dataset consists of 60,000 32x32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images. See the dataset website for more details: https://www.cs.toronto.edu/~kriz/cifar.html.

Do this: Download the dataset from this link (https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz), place it where you want, and unzip it. Then try running the below code to see if you can load the dataset. Change `cifar10_dir` to your data path.

[2]:
```
meta = pickle.load(open('/Users/SamLiang/Documents/Princeton/Classes/Junior␣
↪Semester 1/COS 429/A2/COS429_a2/cifar-10-batches-py/batches.meta', 'rb'),␣
↪encoding='bytes')
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data(cifar10_dir='/
↪Users/SamLiang/Documents/Princeton/Classes/Junior Semester 1/COS 429/A2/
↪COS429_a2/cifar-10-batches-py',
                                                                  ␣
↪num_training=4500,
```

```
                                                                    ␣
↪num_validation=500,
                                                                    num_test=1000)
```

```
Train data shape:  (4500, 32, 32, 3)
Train labels shape:  (4500,)
Validation data shape:  (500, 32, 32, 3)
Validation labels shape:  (500,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[3]:
```
# Visualize a sample image
i = 0
label = y_train[i]
class_name = meta[b'label_names'][label]
plt.imshow(np.uint8(X_train[i])); plt.axis('off')
plt.title('Label: {} ({})'.format(label, class_name)); plt.show()
```

2.2 Question 2. Color features (5 points)

First, we are going to explore using average color features to train a classifier. For each RGB color channel, average the pixel intensities. So a 32x32x3 image will be represented in a 1x3 vector.

Do this: Implement the average color feature. Then train a classifier. Tune the regularization strength to train a good classifier.

[4]:
```
# Freshly load the data
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data(cifar10_dir='/
 ↪Users/SamLiang/Documents/Princeton/Classes/Junior Semester 1/COS 429/A2/
 ↪COS429_a2/cifar-10-batches-py',
                                                                  ␣
 ↪num_training=4500,
                                                                  ␣
 ↪num_validation=500,
                                                                   num_test=1000)
```

```
Train data shape:  (4500, 32, 32, 3)
Train labels shape:  (4500,)
Validation data shape:  (500, 32, 32, 3)
Validation labels shape:  (500,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[5]:
```
# TODO: Compute average color features
start_time = time.time()
X_train = np.mean(np.reshape(X_train, (X_train.shape[0], X_train.shape[1] *␣
 ↪X_train.shape[2], X_train.shape[3])), axis=1)
X_val = np.mean(np.reshape(X_val, (X_val.shape[0], X_val.shape[1] * X_val.
 ↪shape[2], X_val.shape[3])), axis=1)
X_test = np.mean(np.reshape(X_test, (X_test.shape[0], X_test.shape[1] * X_test.
 ↪shape[2], X_test.shape[3])), axis=1)

# TODO: Add bias dimension and transform into columns
X_train = np.hstack((X_train, np.ones((X_train.shape[0], 1))))
X_val = np.hstack((X_val, np.ones((X_val.shape[0], 1))))
X_test = np.hstack((X_test, np.ones((X_test.shape[0], 1))))

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (4500, 4)
Train labels shape:  (4500,)
Validation data shape:  (500, 4)
Validation labels shape:  (500,)
Test data shape:  (1000, 4)
Test labels shape:  (1000,)
```

[6]:
```
# TODO : Define regularization strengths
regularization_strengths = [10**i for i in range(-4, 5)]
```

```
# Train a classifier
softmax_q2 = train(X_train, y_train, X_val, y_val, X_test, y_test,␣
 ↪regularization_strengths)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
reg 1.000000e-04 train accuracy: 0.143778 val accuracy: 0.162000
reg 1.000000e-03 train accuracy: 0.133333 val accuracy: 0.138000
reg 1.000000e-02 train accuracy: 0.120222 val accuracy: 0.126000
reg 1.000000e-01 train accuracy: 0.127111 val accuracy: 0.144000
reg 1.000000e+00 train accuracy: 0.132889 val accuracy: 0.140000
reg 1.000000e+01 train accuracy: 0.124889 val accuracy: 0.134000
reg 1.000000e+02 train accuracy: 0.134444 val accuracy: 0.150000
reg 1.000000e+03 train accuracy: 0.099556 val accuracy: 0.114000
reg 1.000000e+04 train accuracy: 0.099333 val accuracy: 0.116000

best validation accuracy achieved during training: 0.162000

final test set accuracy: 0.147000
--- 3.0278279781341553 seconds ---
```

2.3 Question 3. Bag of SIFT features (15 points)

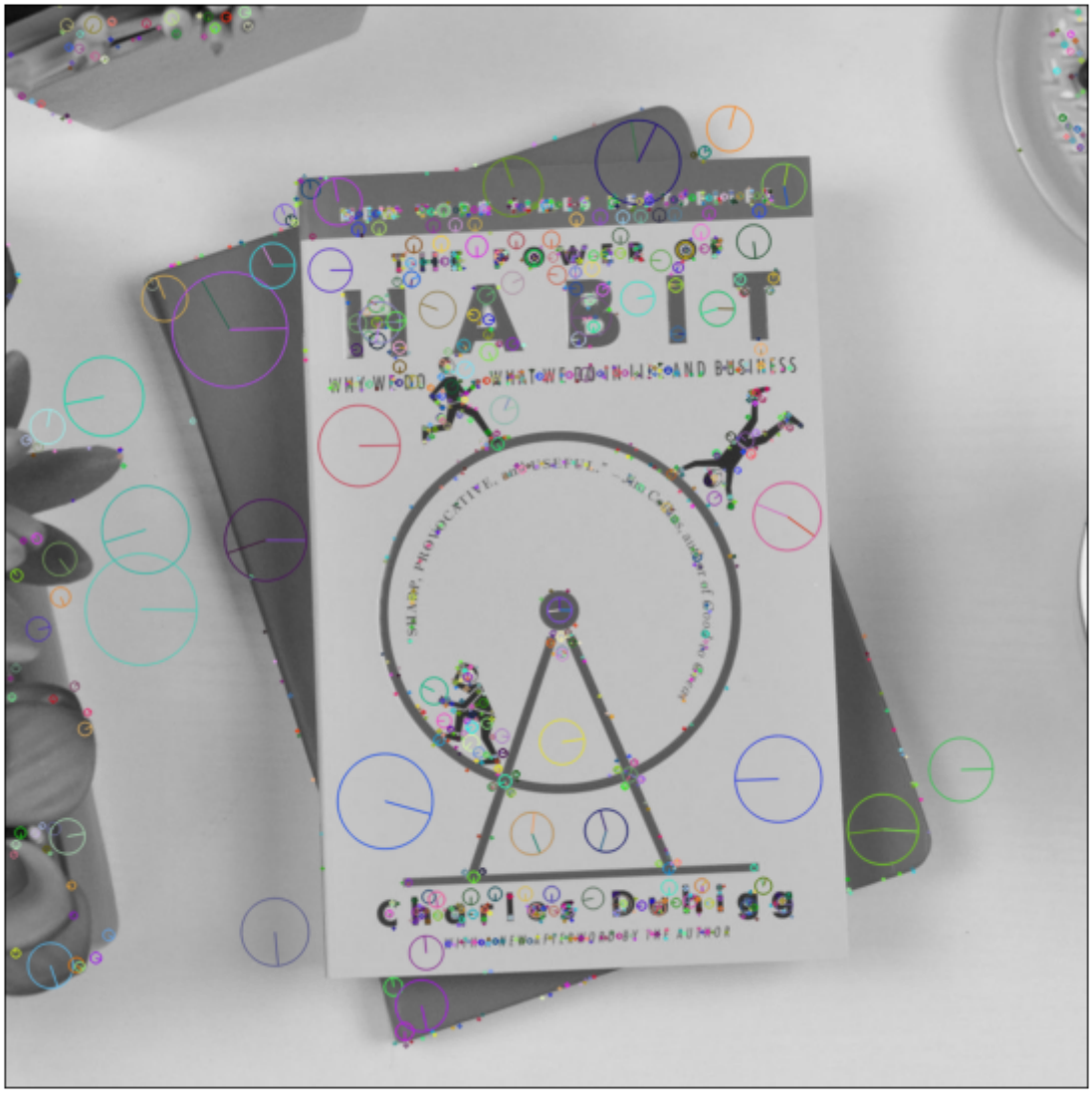
Bag of words models are a popular technique for image classification inspired by models used in natural language processing. The model ignores or downplays word arrangement (spatial information in the image) and classifies based on a histogram of the frequency of visual words. The visual word "vocabulary" is established by clustering a large corpus of local features. In this question, you will extract SIFT features from the training images. These result in a Nx128 dimensional matrix where N is the number of keypoints. After extracting SIFT features from all training images, we can use the K-means clustering algorithm to cluster these features into K clusters each represented by a 128-dimensional centroid. Now we have a bag of visual words (clusters) and can represent each image as a histogram of SIFT features assigned to these clusters. Specifically, each image will be represented as a K-dimensional histogram. Using these representations, you can train a classifier as before.

Do this: Extract SIFT features. Do K-means clustering of the training images' SIFT features. Construct a histogram representation of the images and train a classifier. Specifically, implement `extract_sift()` in `features.py`.

Example for extracting SIFT features Check out OpenCV's tutorial on extracting SIFT features: https://docs.opencv.org/3.4/da/df5/tutorial_py_sift_intro.html.

[7]:
```
# Read in the image
img = cv2.imread('table.jpeg')

# Convert to greyscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

```
# Create a SIFT feature extractor
sift = cv2.xfeatures2d.SIFT_create()

# Detect features from the image
keypoints, descriptors = sift.detectAndCompute(gray, None)

# Draw and visualize the detected keypoints on the image
sift_image = cv2.drawKeypoints(gray, keypoints, img, flags=cv2.
 ↪DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
plt.figure(figsize=(10, 10))
plt.imshow(sift_image)
plt.xticks([]), plt.yticks([])
plt.show()
```



Your work starts here

[8]:
```
# Freshly load the data
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data(cifar10_dir='/
↪Users/SamLiang/Documents/Princeton/Classes/Junior Semester 1/COS 429/A2/
↪COS429_a2/cifar-10-batches-py',
                                                                  ␣
↪num_training=4500,
                                                                  ␣
↪num_validation=500,
                                                                   num_test=1000)
```

```
Train data shape:  (4500, 32, 32, 3)
Train labels shape:  (4500,)
Validation data shape:  (500, 32, 32, 3)
Validation labels shape:  (500,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[9]:
```
# TODO: Write a function to extract sift features
from features import extract_sift_for_dataset
```

[10]:
```
# TODO: Define step_size (sampling density) for sampling keypoints in a grid.
step_size = 4

start_time = time.time()
# TODO: Extract dense SIFT features
X_train_features = np.array(extract_sift_for_dataset(X_train,␣
↪step_size=step_size))
dim = X_train_features.shape
X_train_features_flattened = np.reshape(X_train_features, (dim[0] * dim[1],␣
↪dim[2]))

X_val_features = np.array(extract_sift_for_dataset(X_val, step_size=step_size))
dim = X_val_features.shape
X_val_features_flattened = np.reshape(X_val_features, (dim[0] * dim[1], dim[2]))

X_test_features = np.array(extract_sift_for_dataset(X_test,␣
↪step_size=step_size))
dim = X_test_features.shape
X_test_features_flattened = np.reshape(X_test_features, (dim[0] * dim[1],␣
↪dim[2]))
```

[11]:
```
# TODO: Use your kmeans implementation from part 1 to cluster
# the extracted SIFT features and build a visual vocabulary
# from kmeans import kmeans
from sklearn.cluster import KMeans
```

```
K = 20
niter = 300
# labels_train, centroids = kmeans(X_train_features_flattened, K, niter)
kmeans = KMeans(n_clusters=K, max_iter=niter).fit(X_train_features_flattened)
```

[12]:
```
# TODO: Form histograms for images
# 1. Extract SIFT descriptors from an image
# 2. For each descriptor in the image, find the most similar centroid
# 3. Increase the frequency of that centroid in the histogram for this image
# 4. Normalize the histogram for each image
# 5. Result: N by 7 Matrix where N is the number of images

def hist_creation(X, kmeans, num_of_images, num_of_descriptors, K):
    hist = np.zeros((num_of_images, K)) # number of images by K
    labels = kmeans.predict(X)

    row = 0
    for i in range(len(labels)):
        if i != 0 and i % num_of_descriptors == 0:
#             hist[row] = hist[row]/np.sum(hist[row]) # normalize histogram
            row+=1

        hist[row][labels[i]] += 1

#     hist[row] = hist[row]/np.sum(hist[row]) # normalize final image's␣
 ↪histrogram

    return hist

train_hist = hist_creation(X_train_features_flattened, kmeans, X_train_features.
 ↪shape[0], X_train_features.shape[1], K)
val_hist = hist_creation(X_val_features_flattened, kmeans, X_val_features.
 ↪shape[0], X_val_features.shape[1], K)
test_hist = hist_creation(X_test_features_flattened, kmeans, X_test_features.
 ↪shape[0], X_test_features.shape[1], K)
```

[13]:
```
# TODO: Add bias dimension and transform into columns
X_train = np.hstack((train_hist, np.ones((X_train.shape[0], 1))))
X_val = np.hstack((val_hist, np.ones((X_val.shape[0], 1))))
X_test = np.hstack((test_hist, np.ones((X_test.shape[0], 1))))

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (4500, 21)
Train labels shape:  (4500,)
Validation data shape:  (500, 21)
Validation labels shape:  (500,)
Test data shape:  (1000, 21)
Test labels shape:  (1000,)
```

[14]:
```
# TODO: Define regularization strengths
regularization_strengths = [10 ** i for i in range(-3, 4)]

# Train a classifier
softmax_q3 = train(X_train, y_train, X_val, y_val, X_test, y_test,␣
 ↪regularization_strengths)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
reg 1.000000e-03 train accuracy: 0.137778 val accuracy: 0.126000
reg 1.000000e-02 train accuracy: 0.134667 val accuracy: 0.144000
reg 1.000000e-01 train accuracy: 0.144000 val accuracy: 0.128000
reg 1.000000e+00 train accuracy: 0.137333 val accuracy: 0.144000
reg 1.000000e+01 train accuracy: 0.137556 val accuracy: 0.156000
reg 1.000000e+02 train accuracy: 0.158444 val accuracy: 0.150000
reg 1.000000e+03 train accuracy: 0.249111 val accuracy: 0.274000

best validation accuracy achieved during training: 0.274000

final test set accuracy: 0.257000
--- 112.34253025054932 seconds ---
```

2.4 Question 4. SPM representation (15 points)

One drawback of the bag-of-words approach is that it discards spatial information. Hence, we will now try encoding spatial information using Spatial Pyramid Matching (SPM) proposed in Lazebnik et al. 2006. At a high level, SPM works by breaking up an image into different regions and computing the SIFT descriptor at each region, forming a histogram of visual words in each region, and then concatenatating them into a single 1D vector representation.

Do this: Construct a SPM representation of the images and train a classifier. Specifically, implement `spatial_pyramid_matching()` in `features.py`.

[15]:
```
# Freshly load the data
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data(cifar10_dir='/
 ↪Users/SamLiang/Documents/Princeton/Classes/Junior Semester 1/COS 429/A2/
 ↪COS429_a2/cifar-10-batches-py',
                                                                  ␣
 ↪num_training=4500,
                                                                  ␣
 ↪num_validation=500,
                                                                   num_test=1000)
```

```
Train data shape:  (4500, 32, 32, 3)
Train labels shape:  (4500,)
Validation data shape:  (500, 32, 32, 3)
Validation labels shape:  (500,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[16]:
```
# TODO: Define parameters {L, K, niter}
L = 2 # Number of levels in SPM
K = 30
niter = 150
step_size = 1

start_time = time.time()
# TODO: Extract SIFT features
X_train_features = np.array(extract_sift_for_dataset(X_train,␣
 ↪step_size=step_size))
dim = X_train_features.shape
X_train_features_flattened = np.reshape(X_train_features, (dim[0] * dim[1],␣
 ↪dim[2]))

X_val_features = np.array(extract_sift_for_dataset(X_val, step_size=step_size))
dim = X_val_features.shape
X_val_features_flattened = np.reshape(X_val_features, (dim[0] * dim[1], dim[2]))

X_test_features = np.array(extract_sift_for_dataset(X_test,␣
 ↪step_size=step_size))
dim = X_test_features.shape
X_test_features_flattened = np.reshape(X_test_features, (dim[0] * dim[1],␣
 ↪dim[2]))

# TODO: Use your kmeans implementation from part 1
# to cluster the extracted SIFT features and build a visual vocabulary
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=K, max_iter=niter).fit(X_train_features_flattened)
```

[17]:
```
centroids = kmeans.cluster_centers_
```

[18]:
```
# TODO: Write code to do SPM
from features import spatial_pyramid_matching
```

[19]:
```
# TODO: Get a SPM representation of the extracted SIFT features
# Note: This may take some time to run
X_train_spm = [spatial_pyramid_matching(L,
                                        X_train_features[i].reshape((32, 32,␣
 ↪128)),
```

```
                                        centroids)
              for i in range(len(X_train))]

X_val_spm = [spatial_pyramid_matching(L,
                                      X_val_features[i].reshape((32, 32, 128)),
                                      centroids)
             for i in range(len(X_val))]

X_test_spm = [spatial_pyramid_matching(L,
                                       X_test_features[i].reshape((32, 32,␣
 ↪128)),
                                       centroids)
              for i in range(len(X_test))]

X_train_spm = np.array(X_train_spm)
X_val_spm = np.array(X_val_spm)
X_test_spm = np.array(X_test_spm)
print(X_train_spm.shape, X_val_spm.shape, X_test_spm.shape)
```

```
(4500, 630) (500, 630) (1000, 630)
```

[20]:
```
# TODO: Add bias dimension and transform into columns
X_train = np.hstack((X_train_spm, np.ones((X_train_spm.shape[0], 1))))
X_val = np.hstack((X_val_spm, np.ones((X_val_spm.shape[0], 1))))
X_test = np.hstack((X_test_spm, np.ones((X_test_spm.shape[0], 1))))

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (4500, 631)
Train labels shape:  (4500,)
Validation data shape:  (500, 631)
Validation labels shape:  (500,)
Test data shape:  (1000, 631)
Test labels shape:  (1000,)
```

[21]:
```
# TODO: Define regularization strengths
# regularization_strengths = [10 ** i for i in range(-3, 4)]
regularization_strengths = np.linspace(0, 10000, 100)

# Train a classifier
softmax_q4 = train(X_train, y_train, X_val, y_val, X_test, y_test,␣
 ↪regularization_strengths)
```

```
print("--- %s seconds ---" % (time.time() - start_time))
```

```
reg 0.000000e+00 train accuracy: 0.154667 val accuracy: 0.166000
reg 1.010101e+02 train accuracy: 0.150444 val accuracy: 0.188000
reg 2.020202e+02 train accuracy: 0.185556 val accuracy: 0.202000
reg 3.030303e+02 train accuracy: 0.144000 val accuracy: 0.136000
reg 4.040404e+02 train accuracy: 0.184667 val accuracy: 0.178000
reg 5.050505e+02 train accuracy: 0.189111 val accuracy: 0.158000
reg 6.060606e+02 train accuracy: 0.196889 val accuracy: 0.196000
reg 7.070707e+02 train accuracy: 0.212889 val accuracy: 0.190000
reg 8.080808e+02 train accuracy: 0.255778 val accuracy: 0.258000
reg 9.090909e+02 train accuracy: 0.277778 val accuracy: 0.280000
reg 1.010101e+03 train accuracy: 0.296667 val accuracy: 0.318000
reg 1.111111e+03 train accuracy: 0.312222 val accuracy: 0.312000
reg 1.212121e+03 train accuracy: 0.319111 val accuracy: 0.304000
reg 1.313131e+03 train accuracy: 0.318667 val accuracy: 0.320000
reg 1.414141e+03 train accuracy: 0.321778 val accuracy: 0.320000
reg 1.515152e+03 train accuracy: 0.327778 val accuracy: 0.320000
reg 1.616162e+03 train accuracy: 0.329778 val accuracy: 0.326000
reg 1.717172e+03 train accuracy: 0.330000 val accuracy: 0.318000
reg 1.818182e+03 train accuracy: 0.327333 val accuracy: 0.336000
reg 1.919192e+03 train accuracy: 0.331333 val accuracy: 0.314000
reg 2.020202e+03 train accuracy: 0.332889 val accuracy: 0.340000
reg 2.121212e+03 train accuracy: 0.320444 val accuracy: 0.324000
reg 2.222222e+03 train accuracy: 0.333333 val accuracy: 0.328000
reg 2.323232e+03 train accuracy: 0.329333 val accuracy: 0.324000
reg 2.424242e+03 train accuracy: 0.334889 val accuracy: 0.322000
reg 2.525253e+03 train accuracy: 0.332889 val accuracy: 0.338000
reg 2.626263e+03 train accuracy: 0.325111 val accuracy: 0.332000
reg 2.727273e+03 train accuracy: 0.335556 val accuracy: 0.334000
reg 2.828283e+03 train accuracy: 0.323556 val accuracy: 0.326000
reg 2.929293e+03 train accuracy: 0.330889 val accuracy: 0.332000
reg 3.030303e+03 train accuracy: 0.326222 val accuracy: 0.330000
reg 3.131313e+03 train accuracy: 0.325556 val accuracy: 0.324000
reg 3.232323e+03 train accuracy: 0.330222 val accuracy: 0.320000
reg 3.333333e+03 train accuracy: 0.328667 val accuracy: 0.328000
reg 3.434343e+03 train accuracy: 0.336222 val accuracy: 0.322000
reg 3.535354e+03 train accuracy: 0.332222 val accuracy: 0.316000
reg 3.636364e+03 train accuracy: 0.324667 val accuracy: 0.334000
reg 3.737374e+03 train accuracy: 0.327111 val accuracy: 0.320000
reg 3.838384e+03 train accuracy: 0.327111 val accuracy: 0.318000
reg 3.939394e+03 train accuracy: 0.329778 val accuracy: 0.326000
reg 4.040404e+03 train accuracy: 0.329778 val accuracy: 0.328000
reg 4.141414e+03 train accuracy: 0.326667 val accuracy: 0.332000
reg 4.242424e+03 train accuracy: 0.325333 val accuracy: 0.318000
reg 4.343434e+03 train accuracy: 0.327111 val accuracy: 0.330000
reg 4.444444e+03 train accuracy: 0.328000 val accuracy: 0.324000
```

```
reg 4.545455e+03 train accuracy: 0.332000 val accuracy: 0.342000
reg 4.646465e+03 train accuracy: 0.328444 val accuracy: 0.328000
reg 4.747475e+03 train accuracy: 0.328000 val accuracy: 0.324000
reg 4.848485e+03 train accuracy: 0.336000 val accuracy: 0.326000
reg 4.949495e+03 train accuracy: 0.320667 val accuracy: 0.326000
reg 5.050505e+03 train accuracy: 0.320444 val accuracy: 0.318000
reg 5.151515e+03 train accuracy: 0.327556 val accuracy: 0.324000
reg 5.252525e+03 train accuracy: 0.327333 val accuracy: 0.326000
reg 5.353535e+03 train accuracy: 0.331778 val accuracy: 0.330000
reg 5.454545e+03 train accuracy: 0.317778 val accuracy: 0.318000
reg 5.555556e+03 train accuracy: 0.323333 val accuracy: 0.324000
reg 5.656566e+03 train accuracy: 0.325778 val accuracy: 0.320000
reg 5.757576e+03 train accuracy: 0.330667 val accuracy: 0.316000
reg 5.858586e+03 train accuracy: 0.336000 val accuracy: 0.332000
reg 5.959596e+03 train accuracy: 0.324444 val accuracy: 0.338000
reg 6.060606e+03 train accuracy: 0.324889 val accuracy: 0.328000
reg 6.161616e+03 train accuracy: 0.331111 val accuracy: 0.320000
reg 6.262626e+03 train accuracy: 0.329333 val accuracy: 0.334000
reg 6.363636e+03 train accuracy: 0.333778 val accuracy: 0.336000
reg 6.464646e+03 train accuracy: 0.335778 val accuracy: 0.328000
reg 6.565657e+03 train accuracy: 0.328222 val accuracy: 0.318000
reg 6.666667e+03 train accuracy: 0.321333 val accuracy: 0.306000
reg 6.767677e+03 train accuracy: 0.325111 val accuracy: 0.324000
reg 6.868687e+03 train accuracy: 0.328000 val accuracy: 0.332000
reg 6.969697e+03 train accuracy: 0.322000 val accuracy: 0.328000
reg 7.070707e+03 train accuracy: 0.314667 val accuracy: 0.322000
reg 7.171717e+03 train accuracy: 0.329111 val accuracy: 0.336000
reg 7.272727e+03 train accuracy: 0.325333 val accuracy: 0.328000
reg 7.373737e+03 train accuracy: 0.327111 val accuracy: 0.312000
reg 7.474747e+03 train accuracy: 0.341778 val accuracy: 0.324000
reg 7.575758e+03 train accuracy: 0.330667 val accuracy: 0.324000
reg 7.676768e+03 train accuracy: 0.325111 val accuracy: 0.320000
reg 7.777778e+03 train accuracy: 0.337778 val accuracy: 0.330000
reg 7.878788e+03 train accuracy: 0.327111 val accuracy: 0.326000
reg 7.979798e+03 train accuracy: 0.326667 val accuracy: 0.328000
reg 8.080808e+03 train accuracy: 0.321111 val accuracy: 0.324000
reg 8.181818e+03 train accuracy: 0.326222 val accuracy: 0.330000
reg 8.282828e+03 train accuracy: 0.329333 val accuracy: 0.330000
reg 8.383838e+03 train accuracy: 0.322222 val accuracy: 0.330000
reg 8.484848e+03 train accuracy: 0.338222 val accuracy: 0.326000
reg 8.585859e+03 train accuracy: 0.314889 val accuracy: 0.304000
reg 8.686869e+03 train accuracy: 0.329778 val accuracy: 0.320000
reg 8.787879e+03 train accuracy: 0.330222 val accuracy: 0.318000
reg 8.888889e+03 train accuracy: 0.312889 val accuracy: 0.308000
reg 8.989899e+03 train accuracy: 0.327111 val accuracy: 0.328000
reg 9.090909e+03 train accuracy: 0.323556 val accuracy: 0.306000
reg 9.191919e+03 train accuracy: 0.313333 val accuracy: 0.302000
reg 9.292929e+03 train accuracy: 0.332000 val accuracy: 0.312000
```

```
reg 9.393939e+03 train accuracy: 0.337556 val accuracy: 0.326000
reg 9.494949e+03 train accuracy: 0.328667 val accuracy: 0.310000
reg 9.595960e+03 train accuracy: 0.334444 val accuracy: 0.324000
reg 9.696970e+03 train accuracy: 0.331556 val accuracy: 0.330000
reg 9.797980e+03 train accuracy: 0.336444 val accuracy: 0.322000
reg 9.898990e+03 train accuracy: 0.326000 val accuracy: 0.332000
reg 1.000000e+04 train accuracy: 0.331778 val accuracy: 0.316000

best validation accuracy achieved during training: 0.342000

final test set accuracy: 0.317000
--- 1981.142678976059 seconds ---
```

2.5 Question 5. Histogram of Oriented Gradients (10 points)

Rather than extracting local SIFT features, we can compute a global histogram of oriented gradients (HOG) image descriptor.

Do this: Implement `get_differential_filter()` and `filter_image()` in `features.py`. Then compute HOG descriptors and train a classifier.

[22]:
```
# Freshly load the data
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data(cifar10_dir='/
 ↪Users/SamLiang/Documents/Princeton/Classes/Junior Semester 1/COS 429/A2/
 ↪COS429_a2/cifar-10-batches-py',
                                                                  ␣
 ↪num_training=4500,
                                                                  ␣
 ↪num_validation=500,
                                                                   num_test=1000)
```

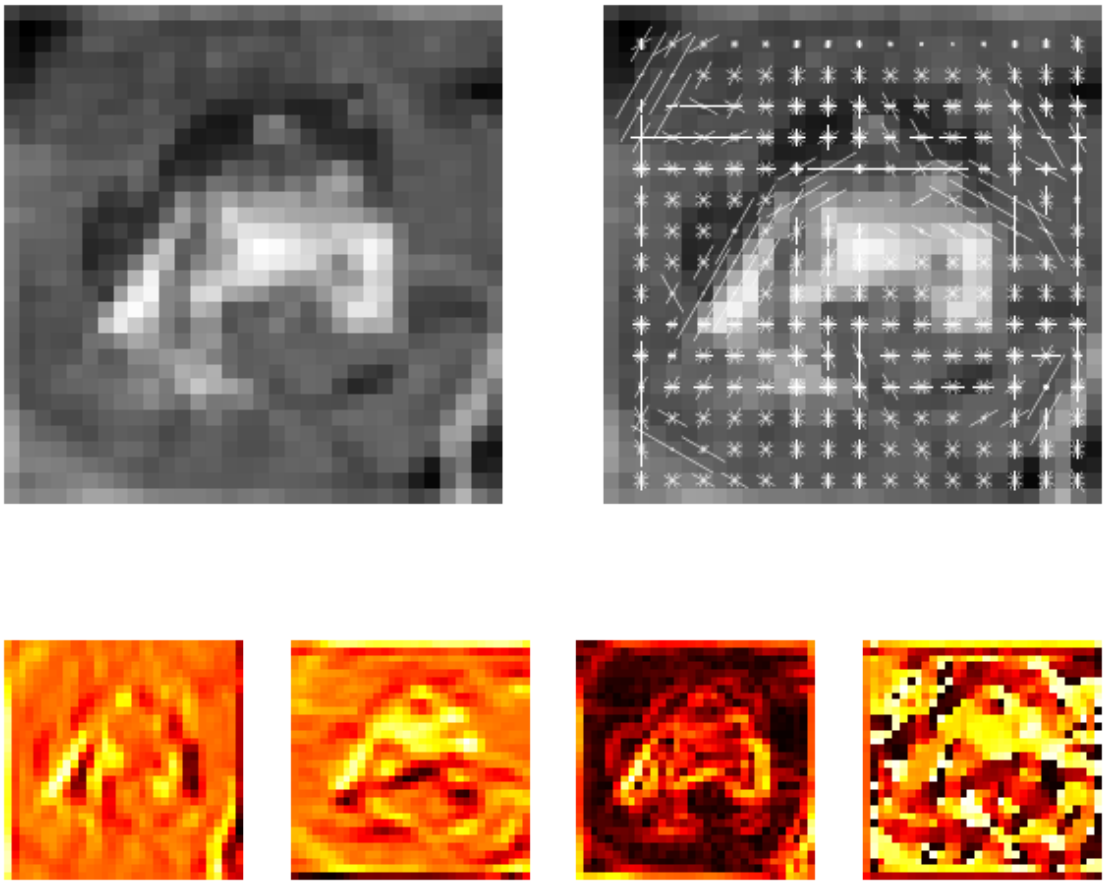
```
Train data shape:  (4500, 32, 32, 3)
Train labels shape:  (4500,)
Validation data shape:  (500, 32, 32, 3)
Validation labels shape:  (500,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[23]:
```
# TODO: Implement get_differential_filter() and filter_image()
# Note: extract_hog() will make use of these two functions.
from features import extract_hog
```

[24]:
```
# TODO: Define parameters
cell_size = 2 # Start with 2 or 4, but feel free to try other parameters
block_size = 2 # Start with 2, but feel free to try other parameters
```

[25]:
```
# Try running your code on a single image
img = X_train[0]
```

```
img = cv2.cvtColor(np.uint8(img), cv2.COLOR_BGR2GRAY)
hog = extract_hog(img, cell_size=cell_size, block_size=block_size, plot=True)
```



[26]:
```
# TODO: Build HOG representations
# Note: This may take some time to run
X_train_hog = []
start_time = time.time()
for img in X_train:
    gray = cv2.cvtColor(np.uint8(img), cv2.COLOR_BGR2GRAY)
    hog = extract_hog(gray, cell_size=cell_size, block_size=block_size)
    X_train_hog.append(hog)

X_train_hog = np.array(X_train_hog)

X_val_hog = []
for img in X_val:
    gray = cv2.cvtColor(np.uint8(img), cv2.COLOR_BGR2GRAY)
    hog = extract_hog(gray, cell_size=cell_size, block_size=block_size)
    X_val_hog.append(hog)
```

```
X_val_hog = np.array(X_val_hog)

X_test_hog = []
for img in X_test:
    gray = cv2.cvtColor(np.uint8(img), cv2.COLOR_BGR2GRAY)
    hog = extract_hog(gray, cell_size=cell_size, block_size=block_size)
    X_test_hog.append(hog)

X_test_hog = np.array(X_test_hog)
```

[27]:
```
# TODO: Add bias dimension and transform into columns
X_train = np.hstack((X_train_hog, np.ones((X_train_hog.shape[0], 1))))
X_val = np.hstack((X_val_hog, np.ones((X_val_hog.shape[0], 1))))
X_test = np.hstack((X_test_hog, np.ones((X_test_hog.shape[0], 1))))

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (4500, 5401)
Train labels shape:  (4500,)
Validation data shape:  (500, 5401)
Validation labels shape:  (500,)
Test data shape:  (1000, 5401)
Test labels shape:  (1000,)
```

[28]:
```
# TODO: Define regularization strengths
regularization_strengths = np.linspace(0, 10000, 30)
regularization_strengths = np.append(regularization_strengths, [10**i for i in␣
 ↪range(-3, 2)])

# Train a classifier
softmax_q5 = train(X_train, y_train, X_val, y_val, X_test, y_test,␣
 ↪regularization_strengths)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
reg 0.000000e+00 train accuracy: 0.176889 val accuracy: 0.204000
reg 1.000000e-03 train accuracy: 0.170444 val accuracy: 0.184000
reg 1.000000e-02 train accuracy: 0.180667 val accuracy: 0.164000
reg 1.000000e-01 train accuracy: 0.191333 val accuracy: 0.208000
reg 1.000000e+00 train accuracy: 0.196222 val accuracy: 0.182000
reg 1.000000e+01 train accuracy: 0.174000 val accuracy: 0.172000
reg 3.448276e+02 train accuracy: 0.156444 val accuracy: 0.170000
```

```
reg 6.896552e+02 train accuracy: 0.135111 val accuracy: 0.144000
reg 1.034483e+03 train accuracy: 0.125333 val accuracy: 0.120000
reg 1.379310e+03 train accuracy: 0.110000 val accuracy: 0.104000
reg 1.724138e+03 train accuracy: 0.105333 val accuracy: 0.100000
reg 2.068966e+03 train accuracy: 0.103333 val accuracy: 0.098000
reg 2.413793e+03 train accuracy: 0.112667 val accuracy: 0.110000
reg 2.758621e+03 train accuracy: 0.103111 val accuracy: 0.098000
reg 3.103448e+03 train accuracy: 0.110444 val accuracy: 0.104000
reg 3.448276e+03 train accuracy: 0.119111 val accuracy: 0.112000
reg 3.793103e+03 train accuracy: 0.104222 val accuracy: 0.098000
reg 4.137931e+03 train accuracy: 0.111333 val accuracy: 0.106000
reg 4.482759e+03 train accuracy: 0.100222 val accuracy: 0.094000
reg 4.827586e+03 train accuracy: 0.103333 val accuracy: 0.098000
reg 5.172414e+03 train accuracy: 0.106222 val accuracy: 0.104000
reg 5.517241e+03 train accuracy: 0.106000 val accuracy: 0.100000
reg 5.862069e+03 train accuracy: 0.101111 val accuracy: 0.094000
reg 6.206897e+03 train accuracy: 0.100222 val accuracy: 0.094000
reg 6.551724e+03 train accuracy: 0.100222 val accuracy: 0.094000
reg 6.896552e+03 train accuracy: 0.102889 val accuracy: 0.098000
reg 7.241379e+03 train accuracy: 0.104000 val accuracy: 0.098000
reg 7.586207e+03 train accuracy: 0.101778 val accuracy: 0.096000
reg 7.931034e+03 train accuracy: 0.107778 val accuracy: 0.102000
reg 8.275862e+03 train accuracy: 0.100889 val accuracy: 0.094000
reg 8.620690e+03 train accuracy: 0.102444 val accuracy: 0.096000
reg 8.965517e+03 train accuracy: 0.100889 val accuracy: 0.094000
reg 9.310345e+03 train accuracy: 0.107111 val accuracy: 0.104000
reg 9.655172e+03 train accuracy: 0.100222 val accuracy: 0.094000
reg 1.000000e+04 train accuracy: 0.100222 val accuracy: 0.094000

best validation accuracy achieved during training: 0.208000

final test set accuracy: 0.185000
--- 493.99778389930725 seconds ---
```

2.6 Question 6. Pixels (5 points)

Finally, let's use the pixels themselves to train a classifier. That is, just reshape a 32x32x3 image into a 32x32x3=3072 vector.

Do this: Process the images and train a classifier.

[29]:
```
# Freshly load the data
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data(cifar10_dir='/
↪Users/SamLiang/Documents/Princeton/Classes/Junior Semester 1/COS 429/A2/
↪COS429_a2/cifar-10-batches-py',

↪num_training=4500,
```

```
                                                                    ␣
↪num_validation=500,
                                                                     num_test=1000)
```

```
Train data shape:  (4500, 32, 32, 3)
Train labels shape:  (4500,)
Validation data shape:  (500, 32, 32, 3)
Validation labels shape:  (500,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[30]:
```
# TODO: Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], 32 * 32 * 3))
X_val = np.reshape(X_val, (X_val.shape[0], 32 * 32 * 3))
X_test = np.reshape(X_test, (X_test.shape[0], 32 * 32 * 3))

start_time = time.time()
# TODO: Normalize the data by subtracting the mean training image from all␣
↪images
mean_train_img = np.mean(X_train, axis=0)
X_train = X_train - mean_train_img
X_val = X_val - mean_train_img
X_test = X_test - mean_train_img

# TODO: Add bias dimension and transform into columns
X_train = np.hstack((X_train, np.ones((X_train.shape[0], 1))))
X_val = np.hstack((X_val, np.ones((X_val.shape[0], 1))))
X_test = np.hstack((X_test, np.ones((X_test.shape[0], 1))))

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (4500, 3073)
Train labels shape:  (4500,)
Validation data shape:  (500, 3073)
Validation labels shape:  (500,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
```

[31]:
```
# TODO: Define regularization strengths
regularization_strengths = np.linspace(0, 10000, 100)
regularization_strengths = np.append(regularization_strengths, [10**i for i in␣
↪range(-3, 2)])
```

```
# Train a classifier
softmax_q6 = train(X_train, y_train, X_val, y_val, X_test, y_test,␣
 ↪regularization_strengths)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
reg 0.000000e+00 train accuracy: 0.470000 val accuracy: 0.302000
reg 1.000000e-03 train accuracy: 0.460889 val accuracy: 0.356000
reg 1.000000e-02 train accuracy: 0.471778 val accuracy: 0.326000
reg 1.000000e-01 train accuracy: 0.467556 val accuracy: 0.318000
reg 1.000000e+00 train accuracy: 0.476222 val accuracy: 0.332000
reg 1.000000e+01 train accuracy: 0.469333 val accuracy: 0.342000
reg 1.010101e+02 train accuracy: 0.494000 val accuracy: 0.336000
reg 2.020202e+02 train accuracy: 0.501556 val accuracy: 0.340000
reg 3.030303e+02 train accuracy: 0.492000 val accuracy: 0.372000
reg 4.040404e+02 train accuracy: 0.482889 val accuracy: 0.364000
reg 5.050505e+02 train accuracy: 0.481778 val accuracy: 0.398000
reg 6.060606e+02 train accuracy: 0.476667 val accuracy: 0.364000
reg 7.070707e+02 train accuracy: 0.466667 val accuracy: 0.372000
reg 8.080808e+02 train accuracy: 0.470667 val accuracy: 0.382000
reg 9.090909e+02 train accuracy: 0.455556 val accuracy: 0.368000
reg 1.010101e+03 train accuracy: 0.452222 val accuracy: 0.342000
reg 1.111111e+03 train accuracy: 0.452000 val accuracy: 0.332000
reg 1.212121e+03 train accuracy: 0.440444 val accuracy: 0.356000
reg 1.313131e+03 train accuracy: 0.448000 val accuracy: 0.390000
reg 1.414141e+03 train accuracy: 0.442444 val accuracy: 0.382000
reg 1.515152e+03 train accuracy: 0.439778 val accuracy: 0.368000
reg 1.616162e+03 train accuracy: 0.442889 val accuracy: 0.388000
reg 1.717172e+03 train accuracy: 0.426444 val accuracy: 0.356000
reg 1.818182e+03 train accuracy: 0.428222 val accuracy: 0.358000
reg 1.919192e+03 train accuracy: 0.434222 val accuracy: 0.372000
reg 2.020202e+03 train accuracy: 0.427556 val accuracy: 0.370000
reg 2.121212e+03 train accuracy: 0.421556 val accuracy: 0.386000
reg 2.222222e+03 train accuracy: 0.421111 val accuracy: 0.360000
reg 2.323232e+03 train accuracy: 0.412000 val accuracy: 0.362000
reg 2.424242e+03 train accuracy: 0.411333 val accuracy: 0.364000
reg 2.525253e+03 train accuracy: 0.423333 val accuracy: 0.378000
reg 2.626263e+03 train accuracy: 0.414000 val accuracy: 0.368000
reg 2.727273e+03 train accuracy: 0.405111 val accuracy: 0.372000
reg 2.828283e+03 train accuracy: 0.412222 val accuracy: 0.400000
reg 2.929293e+03 train accuracy: 0.409333 val accuracy: 0.380000
reg 3.030303e+03 train accuracy: 0.414667 val accuracy: 0.368000
reg 3.131313e+03 train accuracy: 0.399778 val accuracy: 0.348000
reg 3.232323e+03 train accuracy: 0.405333 val accuracy: 0.366000
reg 3.333333e+03 train accuracy: 0.401111 val accuracy: 0.354000
reg 3.434343e+03 train accuracy: 0.408667 val accuracy: 0.362000
reg 3.535354e+03 train accuracy: 0.404889 val accuracy: 0.362000
```

```
reg 3.636364e+03 train accuracy: 0.388667 val accuracy: 0.352000
reg 3.737374e+03 train accuracy: 0.406667 val accuracy: 0.362000
reg 3.838384e+03 train accuracy: 0.401111 val accuracy: 0.374000
reg 3.939394e+03 train accuracy: 0.398444 val accuracy: 0.340000
reg 4.040404e+03 train accuracy: 0.401556 val accuracy: 0.388000
reg 4.141414e+03 train accuracy: 0.402444 val accuracy: 0.372000
reg 4.242424e+03 train accuracy: 0.402667 val accuracy: 0.364000
reg 4.343434e+03 train accuracy: 0.407111 val accuracy: 0.388000
reg 4.444444e+03 train accuracy: 0.395556 val accuracy: 0.356000
reg 4.545455e+03 train accuracy: 0.388222 val accuracy: 0.364000
reg 4.646465e+03 train accuracy: 0.402667 val accuracy: 0.364000
reg 4.747475e+03 train accuracy: 0.392667 val accuracy: 0.318000
reg 4.848485e+03 train accuracy: 0.381111 val accuracy: 0.384000
reg 4.949495e+03 train accuracy: 0.390444 val accuracy: 0.342000
reg 5.050505e+03 train accuracy: 0.397111 val accuracy: 0.354000
reg 5.151515e+03 train accuracy: 0.387778 val accuracy: 0.348000
reg 5.252525e+03 train accuracy: 0.390222 val accuracy: 0.370000
reg 5.353535e+03 train accuracy: 0.396000 val accuracy: 0.368000
reg 5.454545e+03 train accuracy: 0.386000 val accuracy: 0.340000
reg 5.555556e+03 train accuracy: 0.392222 val accuracy: 0.362000
reg 5.656566e+03 train accuracy: 0.396000 val accuracy: 0.390000
reg 5.757576e+03 train accuracy: 0.384000 val accuracy: 0.338000
reg 5.858586e+03 train accuracy: 0.378889 val accuracy: 0.342000
reg 5.959596e+03 train accuracy: 0.395778 val accuracy: 0.344000
reg 6.060606e+03 train accuracy: 0.380667 val accuracy: 0.354000
reg 6.161616e+03 train accuracy: 0.376000 val accuracy: 0.346000
reg 6.262626e+03 train accuracy: 0.382667 val accuracy: 0.352000
reg 6.363636e+03 train accuracy: 0.368222 val accuracy: 0.348000
reg 6.464646e+03 train accuracy: 0.382889 val accuracy: 0.370000
reg 6.565657e+03 train accuracy: 0.368889 val accuracy: 0.346000
reg 6.666667e+03 train accuracy: 0.376000 val accuracy: 0.376000
reg 6.767677e+03 train accuracy: 0.385778 val accuracy: 0.378000
reg 6.868687e+03 train accuracy: 0.383778 val accuracy: 0.372000
reg 6.969697e+03 train accuracy: 0.382889 val accuracy: 0.366000
reg 7.070707e+03 train accuracy: 0.376889 val accuracy: 0.370000
reg 7.171717e+03 train accuracy: 0.371333 val accuracy: 0.332000
reg 7.272727e+03 train accuracy: 0.376000 val accuracy: 0.344000
reg 7.373737e+03 train accuracy: 0.365333 val accuracy: 0.338000
reg 7.474747e+03 train accuracy: 0.369778 val accuracy: 0.364000
reg 7.575758e+03 train accuracy: 0.380222 val accuracy: 0.344000
reg 7.676768e+03 train accuracy: 0.365556 val accuracy: 0.326000
reg 7.777778e+03 train accuracy: 0.366222 val accuracy: 0.366000
reg 7.878788e+03 train accuracy: 0.382222 val accuracy: 0.366000
reg 7.979798e+03 train accuracy: 0.358667 val accuracy: 0.352000
reg 8.080808e+03 train accuracy: 0.366889 val accuracy: 0.348000
reg 8.181818e+03 train accuracy: 0.370444 val accuracy: 0.326000
reg 8.282828e+03 train accuracy: 0.361778 val accuracy: 0.318000
reg 8.383838e+03 train accuracy: 0.358444 val accuracy: 0.310000
```

```
reg 8.484848e+03 train accuracy: 0.363333 val accuracy: 0.318000
reg 8.585859e+03 train accuracy: 0.379778 val accuracy: 0.342000
reg 8.686869e+03 train accuracy: 0.371333 val accuracy: 0.352000
reg 8.787879e+03 train accuracy: 0.339556 val accuracy: 0.344000
reg 8.888889e+03 train accuracy: 0.369556 val accuracy: 0.328000
reg 8.989899e+03 train accuracy: 0.379333 val accuracy: 0.346000
reg 9.090909e+03 train accuracy: 0.372000 val accuracy: 0.346000
reg 9.191919e+03 train accuracy: 0.360889 val accuracy: 0.330000
reg 9.292929e+03 train accuracy: 0.361556 val accuracy: 0.354000
reg 9.393939e+03 train accuracy: 0.358222 val accuracy: 0.330000
reg 9.494949e+03 train accuracy: 0.364889 val accuracy: 0.350000
reg 9.595960e+03 train accuracy: 0.370889 val accuracy: 0.374000
reg 9.696970e+03 train accuracy: 0.366222 val accuracy: 0.318000
reg 9.797980e+03 train accuracy: 0.359333 val accuracy: 0.374000
reg 9.898990e+03 train accuracy: 0.374667 val accuracy: 0.372000
reg 1.000000e+04 train accuracy: 0.371778 val accuracy: 0.356000

best validation accuracy achieved during training: 0.400000

final test set accuracy: 0.361000
--- 384.78232979774475 seconds ---
```

2.7 Question 7. Results (10 points)

Do this:

7-a. Create a table of the five models' achieved accuracy, best hyperparameter, and runtime. (6 points)

7-b. Briefly describe your results in a few sentences. Feel free to share your experience and highlight any interesting observations (e.g., you had to do more hyperparameter tuning for some than others). (4 points)

Model	Accuracy	Best Hyperparameter	Runtime (seconds)
Color Features	0.158	$\lambda = 1$	2.999
Bag of SIFT features	0.257	K = 20, $\lambda = 1000$, step_size = 4	112.342
SPM	0.317	K = 30, L=2, $\lambda = 4545.455$, step_size = 1	1981.143
HOG	0.185	$\lambda = 0.1$, block_size = 2, cell_size = 2	493.998
Pixels	0.361	$\lambda = 2828.283$	384.782

7b. The optimization process took us some time. However, we experienced that in SPM going beyond L = 2 did not provide better performance. Having K in the order of 100s was also did not contribute much to the accuracy. We also saw that normalization affected our results.

2.8 Question 8. Analysis (10 points)

Do this: Create a confusion matrix for each of the five models. Feel free to use existing implementations such as sklearn.metrics.confusion_matrix but make sure to interpret some subset of

the results and demonstrate that you understand what the values in the confusion matrices mean. Do the confusion matrices reveal any interesting insights (e.g., truck is always misclassified as automobile)? For each of the 10 classes, which model works best? Describe any hypotheses you have on the results. One or two paragraphs would be sufficient.

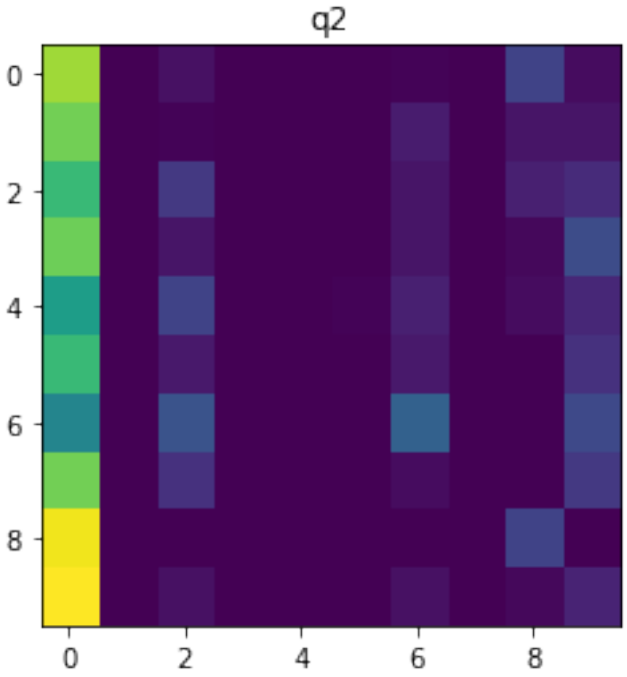
[32]:
```
# Freshly load the data
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data(cifar10_dir='/
 ↪Users/SamLiang/Documents/Princeton/Classes/Junior Semester 1/COS 429/A2/
 ↪COS429_a2/cifar-10-batches-py',
                                                                  ␣
 ↪num_training=4500,
                                                                  ␣
 ↪num_validation=500,
                                                                   num_test=1000)
```

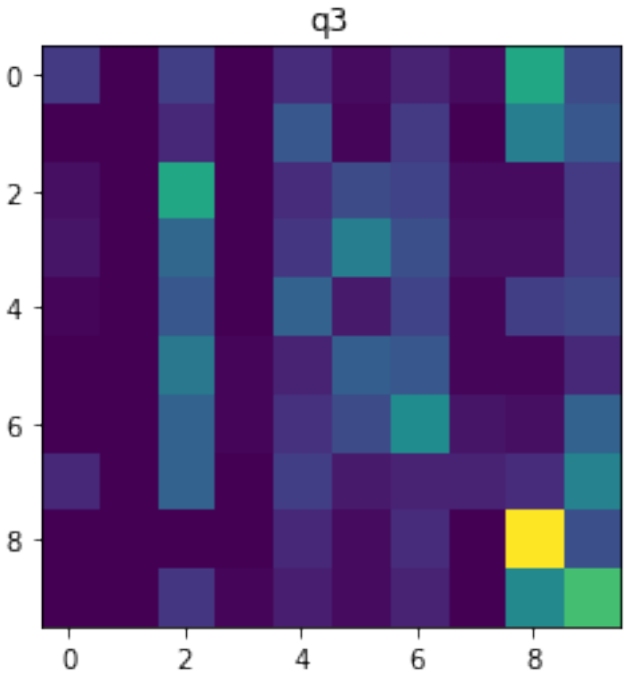
```
Train data shape:  (4500, 32, 32, 3)
Train labels shape:  (4500,)
Validation data shape:  (500, 32, 32, 3)
Validation labels shape:  (500,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

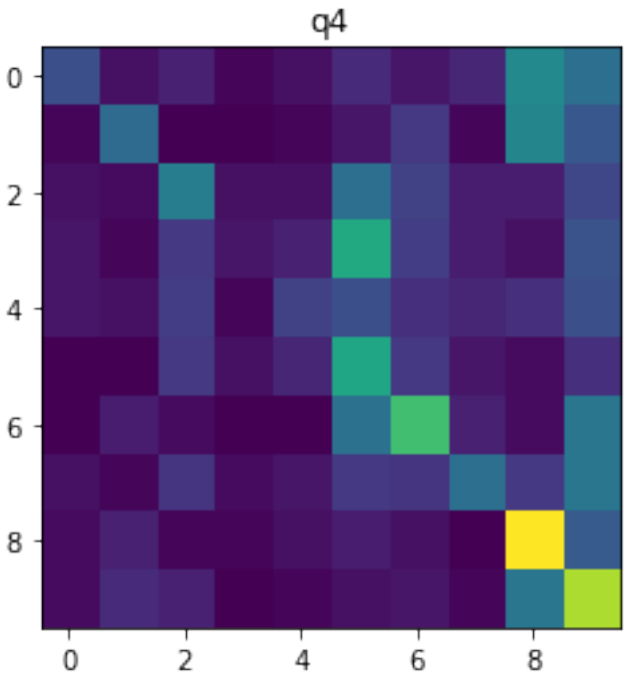
[33]:
```
from sklearn.metrics import confusion_matrix
```

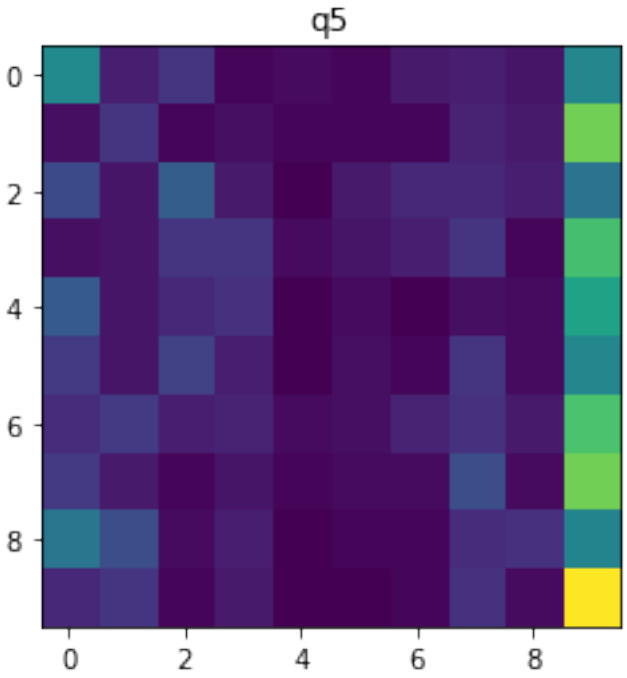
[46]:
```
# Extract the features that will be used as an input to the model.

# q2 - color
X_test_q2 = np.mean(np.reshape(X_test, (X_test.shape[0], X_test.shape[1] *␣
 ↪X_test.shape[2], X_test.shape[3])), axis=1)
X_test_q2 = np.hstack((X_test_q2, np.ones((X_test_q2.shape[0], 1))))

# q3 - bag of words
X_test_q3 = np.hstack((test_hist, np.ones((X_test.shape[0], 1))))

# q4 - spm
X_test_q4 = np.hstack((X_test_spm, np.ones((X_test_spm.shape[0], 1))))

# q5 - hog
X_test_q5 = np.hstack((X_test_hog, np.ones((X_test_hog.shape[0], 1))))

# q6 - pixel
X_test_q6 = np.reshape(X_test, (X_test.shape[0], 32 * 32 * 3))
X_test_q6 = X_test_q6 - mean_train_img
X_test_q6 = np.hstack((X_test_q6, np.ones((X_test_q6.shape[0], 1))))

# Evaluating the predictions
y_test_pred_q2 = softmax_q2.predict(X_test_q2) # color
```

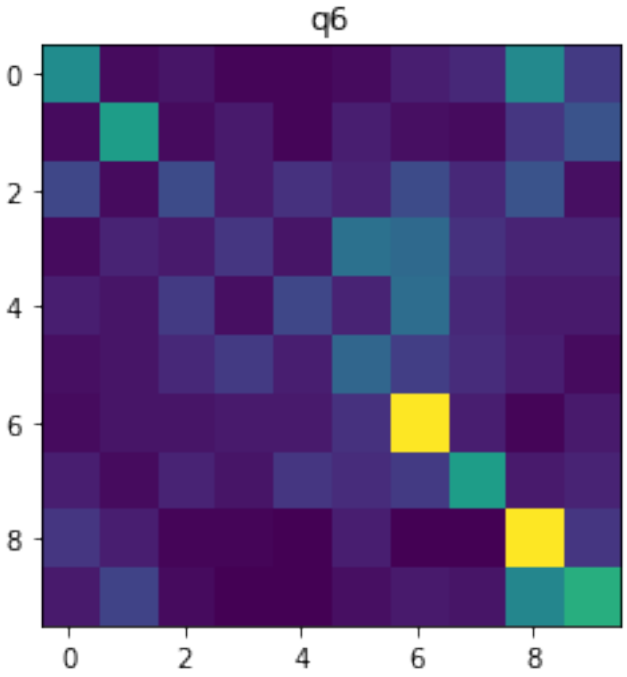
```
y_test_pred_q3 = softmax_q3.predict(X_test_q3) # bag of words
y_test_pred_q4 = softmax_q4.predict(X_test_q4) # spm
y_test_pred_q5 = softmax_q5.predict(X_test_q5) # hog
y_test_pred_q6 = softmax_q6.predict(X_test_q6) # pixel

# confusion matrices
conf_matr_q2 = confusion_matrix(y_test, y_test_pred_q2)
conf_matr_q3 = confusion_matrix(y_test, y_test_pred_q3)
conf_matr_q4 = confusion_matrix(y_test, y_test_pred_q4)
conf_matr_q5 = confusion_matrix(y_test, y_test_pred_q5)
conf_matr_q6 = confusion_matrix(y_test, y_test_pred_q6)

# plot
plt.imshow(conf_matr_q2)
plt.title("q2")
plt.show()
plt.imshow(conf_matr_q3)
plt.title("q3")
plt.show()
plt.imshow(conf_matr_q4)
plt.title("q4")
plt.show()
plt.imshow(conf_matr_q5)
plt.title("q5")
plt.show()
plt.imshow(conf_matr_q6)
plt.title("q6")
plt.show()
```


q2
0
2
4
6
8
0
2
4
6
8


q3
0
2
4
6
8
0
2
4
6
8


q4
0
2
4
6
8
0
2
4
6
8

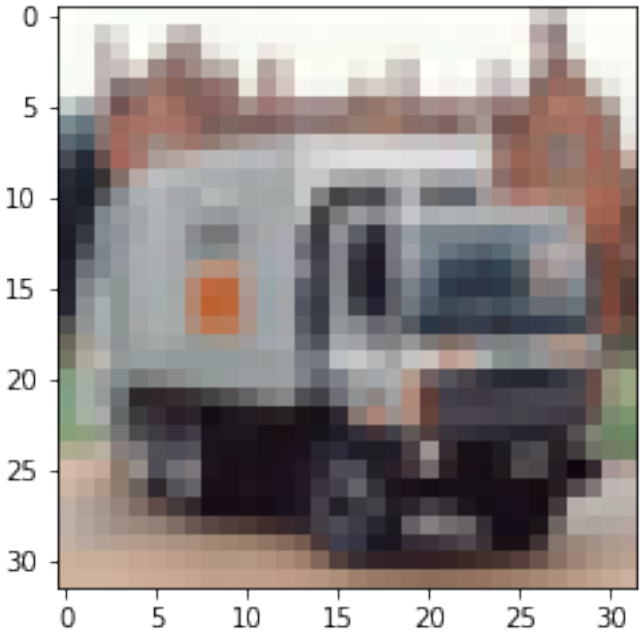

q5
0
2
4
6
8
0
2
4
6
8

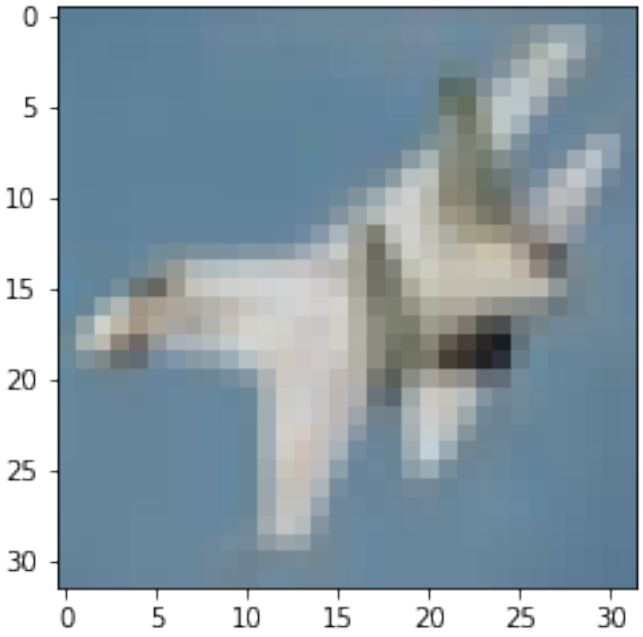


Comments on the confusion matrix: In the Q2 model, we realize that no matter what the actual class is, the model predicts class 0 (which corresponds to the airplane class). In the Q5 model, similarly we also realize that the model predicts class 9 (which corresponds to the trucks) no matter what the true class is. In the other models, we have a diagonal distribution that we more expect from the confusion matrix. In general, we noted that for the classification of classes, Q4 and Q6 worked better (i.e., the confusion matrix is more close to being diagonal). This is expected from the relatively high value of their testing accuracy we reported at the end of their respective questions.

[53]:
```
print(y_test[:20]) # finding some class representatives for demonstration␣
↪purposes.
plt.imshow(X_test[11]/255.0) # class 9
plt.show()
plt.imshow(X_test[10]/255.0) # class 0
plt.show()
```

```
[3 8 8 0 6 6 1 6 3 1 0 9 5 7 9 8 5 7 8 6]
```





2.9 Question 9. Improvement (9 points)

Do this: Identify one shortcoming of one or few of the systems you've worked with. Name an improvement you can implement to improve the system(s). You don't have to actually implement

your proposed improvement, but describe exactly how you could go about implementing it and what pitfalls you might anticipate. What would be the pros and cons of this intervention? One or two paragraphs would be sufficient.

Answer: We'll describe a few shortcomings that we identified. First, as the instructions, hard-coded values, and further deliberations on Ed suggests; in our dense feature extraction for SPM we assumed step_size=1. In order to handle more general step_size cases, we need to pass in the size of the images as well (because the features parameter won't have the same spatial dimensions as the image). Moreover, the looping over features will change as the dimensions do not match anymore. We anticipate difficulties in changing the functional interface and handling the custom looping over image dimensions but for the feature placement in blocks.

Second shortcoming that we identified is that Bag of Words does not take into account spatial structure. Thus, for example, if there is a blue sky at the top and sand at the bottom, the classifier should rank that this is a beach with a high probability. But, bag of words discards spatial structure in the image. An improvement is to use SPM with bag of words. That way we can somewhat perform geometric corresponce between images and take into account spatial structure. We would implement this by using extracting histograms of descriptors on each block at each level in the spatial pyramid.

Another shortcoming is that models that take into account spacial structure like SPM would not work well on categories that have a lot of intraclass variation because then there is little hallmark features to a scene that makes scenes/images in these categories easily recognizable. An improvement is to use neural networks as that would allow the model to learn deeper features on the image than possible with the current models. We would probably use a CNN to implement this neural network. A potential pitfall is that it is complex so implementing all the layers and doing the gradients would be complicated and more error prone. The pros of this intervention is the model will learn deeper features of the image and be more robust to images with high variation. The cons of this intervention is that the neural network is not as interpretable and it requires a lot of training data.

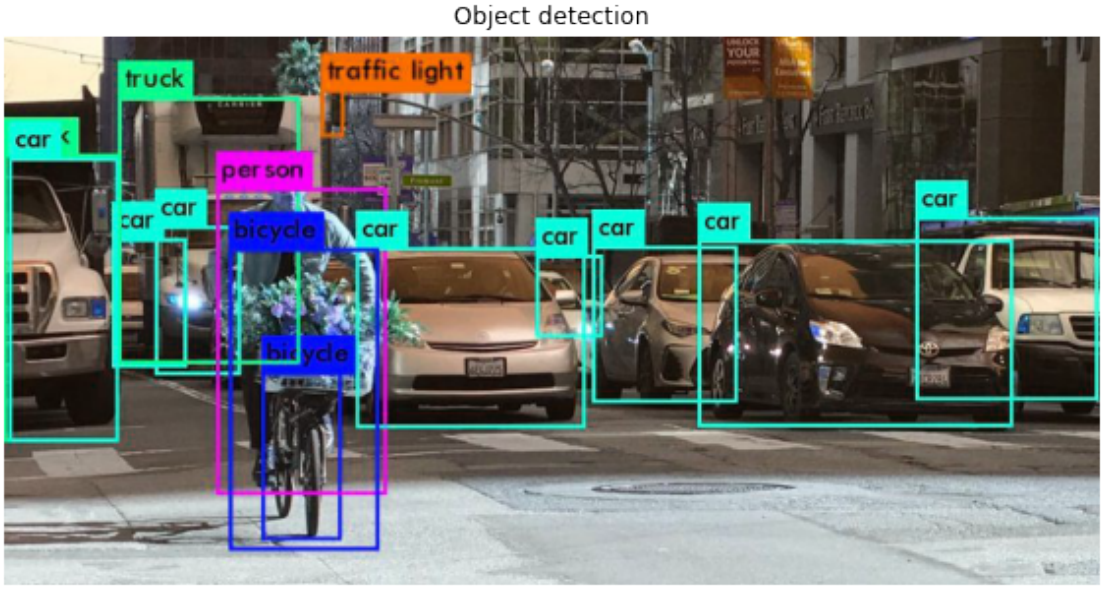
2.10 Question 10. What to Use (6 points)

So far we explored how different features work for 10-way image classification.

Do this:

9-a. For the task of **object detection**, which features do you think would work best? Describe your reasons in a few senteces. (2 points)

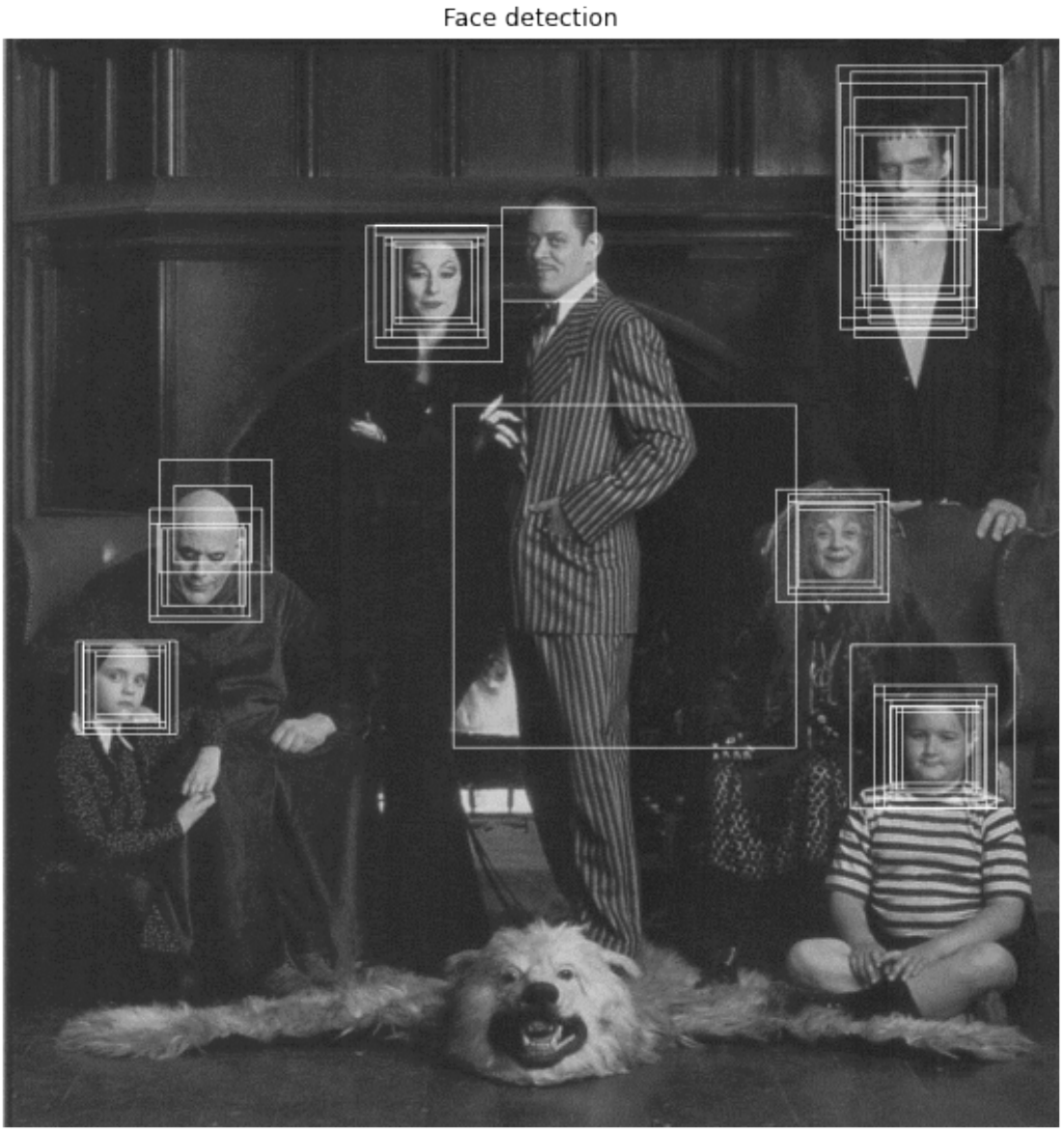
[54]:
```
img = cv2.imread('objectdetection.jpeg')
plt.figure(figsize=(10, 10)); plt.title('Object detection')
plt.imshow(img); plt.axis('off'); plt.show()
```



Answer to 9a: In order to successful object detection, one elementary way is HoG features because unlike the other models, they provide a template and outline for the objects (as they are commonly used in pedestrian detection). Furthermore, better features would be using HoG with part/spring-based models to generate parts features and springs features for further generalization.

9-b. For the task of **face detection**, which features do you think would work best? Describe your reasons in a few senteces. (2 points)

[55]:
```
img = cv2.imread('facedetection.png')
plt.figure(figsize=(10, 10)); plt.title('Face detection')
plt.imshow(img); plt.axis('off'); plt.show()
```

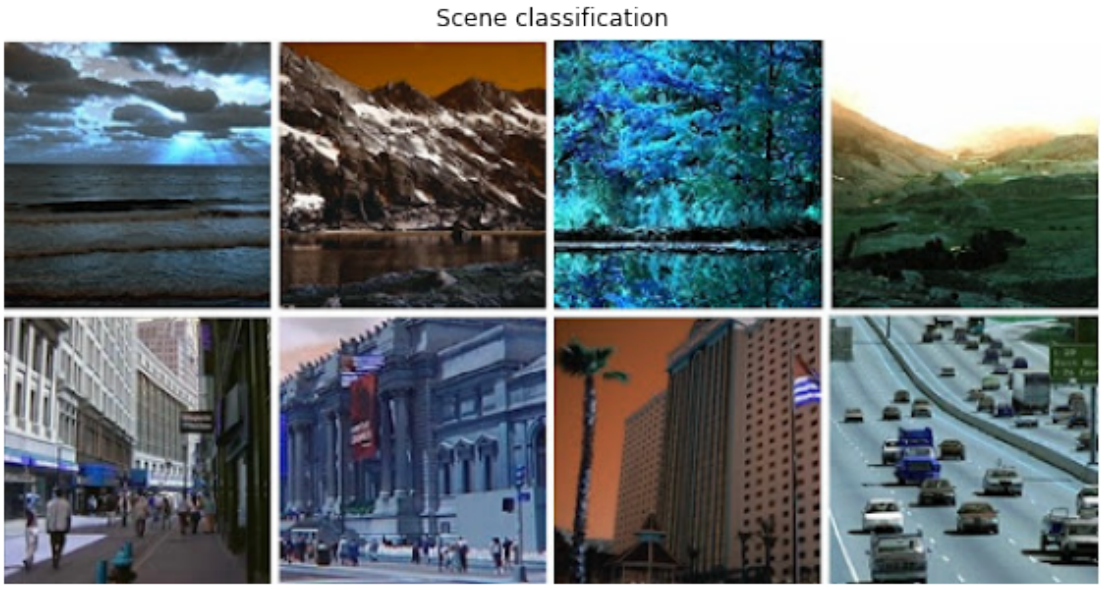


Answer to 9b: For face detection, we'll use HoG features because we need to locate the face in the image. Just using SPM or BoW will not work adequately to match the defining features for faces, also HoG performs well in the structured images and faces are well structured. HoG can also detect the faces from the background as well. Note that using SIFT would work as well because we can easily build the conceptbook / histogram of face-defining features.

9-c. For the task of **scene classification**, which features do you think would work best? Describe your reasons in a few senteces. (2 points)

[56]:
```
img = cv2.imread('sceneclassification.jpeg')
plt.figure(figsize=(10, 10)); plt.title('Scene classification')
```

```
plt.imshow(img); plt.axis('off'); plt.show()
```



Answer to 9c: As we previously briefly mentioned, we want the spatial awareness in the model for scene classification as most scenes have some common prominent features in certain spatial locations (e.g., blue sky over yellow sand). Therefore, we pick the local spatial-aware SPM for scene classification.