CSci 4270 and 6270
Computational Vision,
Fall Semester, 2019
Homework 6
Due: Tuesday, November 26 at 5 pm

Overview — Please Read Carefully

All students must complete Part 1 of this assignment. Students in the graduate version of the class, 6270, must complete Part 2 as well.

The focus of this assignment is scene classification. In particular you will write classifiers to determine the "background" class of a scene. The five possibilities we will consider are grass, wheat field, road, ocean and red carpet. Some of these are relatively easy, but others are hard. A zip file containing the images can be found at

```
https://drive.google.com/open?id=134gx0iS9pcI2P_S7vduPoQ79e3uOHNpQ
```

In Problem 1 you will implement a descriptor and a linear SVM to classify the scene, while in Problem 2 you will implement two neural networks.

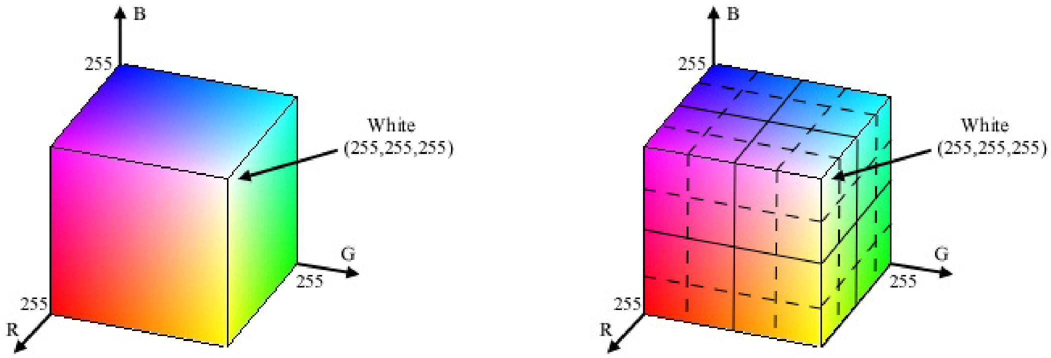
1. (**60 points**) In our lecture on detection we focused on the "HoG" — histogram of oriented gradients — descriptor. After it was published, many different types of descriptors were invented for many applications. For this problem you are going to implement a descriptor that combines location and color. It will include no gradient information. One long descriptor vector will be computed for each image, and then a series of SVM classifiers will be applied to the descriptor vector to make a decision. We strongly urge you to write two scripts to solve this problem, one to compute and save the descriptors for each image, and one to reload the descriptors, train the classifiers, and evaluate the performance.

   The descriptor is formed from an image by computing a 3d color histogram in each of a series of overlapping subimages, unraveling each histogram into a vector (1d NumPy array), and concatenating the resulting vectors into one long vector. The key parameter here will be the value of t, the number of histogram bins in each of the three color dimensions. Fortunately, calculation of color histograms is straightforward. Here is example code to form an image of random values and compute its 3d color histogram:

```
import numpy as np

#  Generate a small image of random values.
M, N = 20, 24
img = np.floor(256 * np.random.random(M * N * 3)).astype(np.uint8)

# Calculate and print the 3d histogram.
t = 4
pixels = img.reshape(M*N, 3)
hist, _ = np.histogramdd(pixels, (t, t, t))
print(’histogram shape:’, hist.shape)   # should be t, t, t
print(’histogram:’, hist)
print(np.sum(hist))                     # should sum to M*N
```



(a)

(b)

Figure 1: Color histogram bins

Each histogram bin covers a range of $u = 256/t$ gray levels in each color dimension. For example, `hist[i, j, k]` is the number of pixels in the random image whose red value is in the range $[i*u, (i+1)*u)$ **and** whose green value is in the range $[j*u, (j+1)*u)$ **and** whose blue value is in the range $[k*u, (k+1)*u)$. Figure 1 illustrates the partitioning of the RGB color cube into $4 \times 4 \times 4$ regions.

As mentioned above, the image will be divided into overlapping subimage blocks and one color histogram size of t^3 will be computed in each block. These will be concatenated to form the final histogram. Let b_w be the number of subimage blocks across the image and let b_h be the number of blocks going down the image. This will produce $b_w \cdot b_h$ blocks overall, and a final descriptor vector of size $t^3 \cdot b_w \cdot b_h$. To compute the blocks in an image of $H \times W$ pixels, let

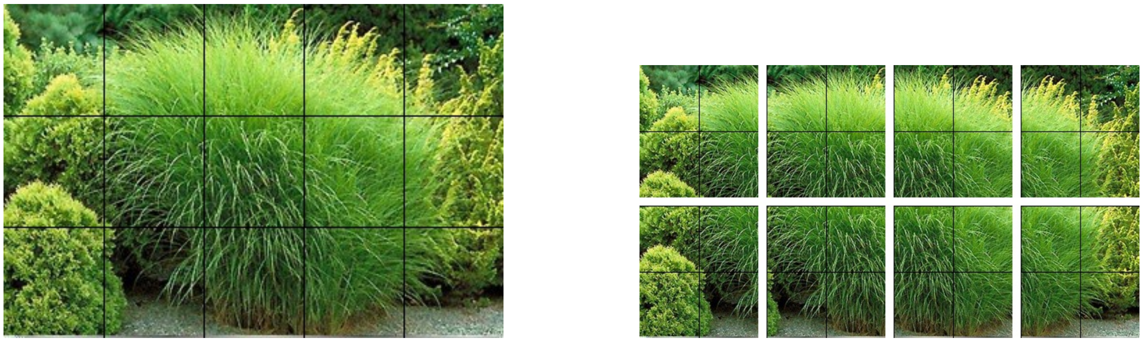
$$\Delta_w = \frac{W}{b_w + 1} \qquad \text{and} \qquad \Delta_h = \frac{H}{b_h + 1}.$$

The blocks will each cover $2\Delta_w$ columns and $2\Delta_h$ rows of pixels. The image pixel positions of the upper left corners of the blocks will be

$$(m\Delta_w, n\Delta_h), \qquad \text{for } m = 0, \ldots, b_w - 1,\ n = 0, \ldots, b_h - 1.$$

Note that some pixels will contribute to only one histogram, some will contribute to two, and others will contribute four. (The same is true of the HoG descriptor.) Figure 2 illustrates the formation of blocks.

After you have computed the descriptors, you will train a series of SVM classifiers, one for each class. To do so, you will be given a set of 4000 training images, $\{I_i\}$, with class labels $y_i \in (1, \ldots, k)$ (for us, $k = 5$). To train classifier C_j, images with label $y_i = j$ are treated as $y_i = +1$ in linear SVM training and images with label $y_i \neq j$ are treated as $y_i = -1$. This will be repeated for each of the k classifiers. The descriptor is computed for each training image I_i to form the data vectors $\mathbf{x}_i$.



Original image with Δ_w and Δ_h spaced lines   Blocks of pixels over which histograms are formed

Figure 2: Image block tiling for $b_w = 4$ and $b_h = 2$.

Each resulting classifier C_j will have a weight vector $\mathbf{w}_j$ and offset b_j. The score for classifier j for a test image with descriptor vector $\mathbf{x}$ is

$$d_j = \frac{1}{\|\mathbf{w}_j\|}\left[\mathbf{w}_j^\top \mathbf{x} + b_j\right].$$

(Recall that the $1/\|\mathbf{w}_j\|$ ensures that d_j is a signed distance.) The classification for the test image I is the class associated with the value of j that gives the maximum d_j score. This is used even if none of the d_j scores are positive.

For each SVM classifier, output simple statistics on the training set (the "training error") including just the percentage (or fraction) of errors for each.

After complete training, you will test your classifiers with the set of 1000 test images. Each will be run as described above and the label will be compared to the known correct label. You will output the percentage correct for each category, followed by a $k \times k$ confusion matrix. The confusion matrix entry at row r and column c shows the number of times when r was the correct class label and c was the chosen class label. The confusion matrix would have all 0's in the non-diagonal entries when the SVM classifier is operating at 100% accuracy.

Some Details

(a) It is very important that you use `histogramdd` or a similar function to compute the histogram. Do not iterate over the pixels in the image using for loops: you will have serious trouble handling the volume of images. You can iterate over the indices of the subimage blocks and then use Numpy calls as above to form the histograms.

(b) As mentioned above, we suggest that you write one script to compute and save the descriptors for all training and test images, and then write a second script to train your SVM classifiers and generate your test output. We suggest that you use Python's `pickle` module.

(c) For your SVM implementation, we suggest using `sklearn.svm.LinearSVC`. To use the scikit-learn (sklearn) Python module, you will need to install the package. If you are using Anaconda, as suggested earlier in the semester, you can simply run

```
conda install scikit-learn
```

(d) When training your model, **do not** run your code on the test images until you have completed your training, including parameter tuning. The LinearSVC object can be used with all of the default settings except for the error tuning parameter C. Play with different values of C (the cost of points being in the SVM's margin, as discussed in lecture) to optimize your SVM's performance for each class. The expected values of C will fall in the range $[0.1, 10.0]$ with performances varying across classes. You can use a different value of C for each class, and you should report these values.

(e) The confusion matrix can be made using Matplotlib or `sklearn.metrics.confusion_matrix`

(f) The computation of feature extraction might still be time consuming even with efficient Numpy use. We suggest that you develop and debug your program using a subset of the training image set before running your final version on the full training and test sets.

(g) We suggest using at a minimum $t = 4$ and $b_w = b_h = 4$. This will give a descriptor of size 1,024.

(h) Finally, you can investigate the addition of gradient histograms to your descriptors. Doing so, with a careful analysis of the impacts, can earn you up to 5 points extra credit.

Submit your code, your output from your final training and test runs, and a write-up. This should contain a description of any design choices and parameter settings, along with a summary discussion of when your classifier works well, when it works poorly, and why.

2. (**60 points**) We continue the background classification problem using neural networks. Specifically, you will use *pytorch* to implement two neural networks, one using only fully-connected layers, and the other using convolutional layers in addition to fully-connected layers. The networks will each start directly with the input images so you will not need to write or use any code to do manual preprocessing or descriptor formation. Therefore, once you understand how to use pytorch the code you actually write will in fact be quite short. Your output should be similar in style to your output from Problem 1.

   Make sure that your write-up includes a discussion of the design of your networks and why you made those choices.

   To help you get started I've provided a Jupyter notebook (on the Piazza site) that illustrates some of the main concepts of pytorch, starting with Tensors and Variables and proceeding to networks, loss functions, and optimizers. This also includes pointers to tutorials on `pytorch.org`. This notebook will be discussed in class on November 14 and 18. If you already know TensorFlow, you'll find pytorch quite straightforward.

   Two side notes:

   (a) PyTorch includes some excellent tools for uploading and transforming images into Tensor objects. For this assignment, you will not need to use these since you've already written code for image input for the previous problem that gathers images into numpy arrays and it is trivial to transform these objects into pytorch tensors.

   (b) Because of the size of the images, you might find it quite computationally expensive and tedious to use a fully-connected network on the full-sized images, especially if you don't have a usable GPU on your computer. Therefore, you are welcome to resize the images before input to your first network. In fact, we strongly suggest that you start with significantly downsized images first and then see how far you can scale up!

Part 2 — Grad Students (6270) Only

1. (**10 points**) Consider a non-convolutional neural network that starts from an RGB input image of size $N \times N$. It has L layers with h nodes per hidden layer, and n nodes in the output layer. Suppose each layer is fully-connected to the previous layer and there is a separate bias term at each node of at each layer. Derive a formula to describe the number of learnable parameters in the network.

2. (**10 points**) Consider a convolutional neural network applied to an RGB input image of size $N \times N$ where, for simplicity of analysis, N is a power of 2. Suppose that

   - the convolutions each cover $k \times k$ pixels,
   - there are d different convolutions per convolution layer,
   - padding is used so that convolutions do not results in image shrinkage, and
   - after each convolution layer there is a max pooling layer applied over non-overlapping 2×2 pixel regions.

   Suppose also that there L convolution layers, followed by F fully-connected layers with h nodes per layer, and $n^{(o)}$ nodes in the output layer. Derive an expression for the number of learnable convolution parameters and the number of learnable parameters in the fully-connected and output layers.

3. (**20 points**) So far, in the output layer of our networks we have used the same activation function as we did in the hidden layers and then applied a mean square error loss (cost) function to evaluate the output. More commonly, however, the output layer uses a special activation function called the *softmax* that forces the output to be a probability distribution (non-negative values that sum to 1), and this is combined with the *cross entropy loss function* to generate the error signals that start backpropagation. To be specific, let

   $$z_j^{(L)} = \sum_{k=1}^{n^{(L-1)}} w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)}$$

   be the input to the node j at the last layer, layer (L). Notationally, we drop the (L) superscript and just write z_j. Also, we use the notation p_j as the output from node j in the output layer instead of $a_j^{(L)}$, both to simplify and to indicate that the output is, mathematically, a probability. Using these two notational changes, the softmax activation output is defined as

   $$p_j = \frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}}.$$

   (where we've adopted the short-hand n for $n^{(L)}$) and the cross-entropy loss function for expected binary output vector $\mathbf{y}$ is

   $$L(\mathbf{p}, \mathbf{y}) = -\sum_i y_i \log p_i.$$

   (a) Show that the activations across the output layer truly form a probability distribution.

   (b) Show that the derivative of p_i with respect to z_k is

   $$\frac{\partial p_i}{\partial z_k} = p_i(\delta_{ik} - p_k).$$

   where δ_{ik} is the Kronecker delta function.

(c) Use this to show that the derivative of L with respect to the input at the i node at the output layer has the amazingly simple, elegant form

$$\frac{\partial L}{\partial z_i} = p_i - y_i$$