Carnegie Mellon University
16-720: Computer Vision
Histograms of Oriented Gradient (HOG) Image Classification

- **Due date.** Please refer to course schedule for the due date for **HW1**.
- **Gradescope submission.** You will need to submit both (1) a `YourAndrewName.pdf` and (2) a `YourAndrewName.zip` file containing your code, either as standalone python (`.py` or colab notebooks (`.ipynb`). We suggest you create a PDF by `print`ing your colab notebook from your browser. However, this can improperly cutoff images that straddle multiple pages. In this case, we suggest you download such images separately and explicitly append them to your PDF using online tools such as https://combinepdf.com/. You may also find it useful to look at this post: https://askubuntu.com/questions/2799/how-to-merge-several-pdf-files. Remember to use Gradescope's functionality to mark pages that provide answers for specific questions, such as code or visualizations.
- **Acknowledgements.** This homework is based on earlier versions created by Andrew Owens, Fei-Fei Li, Justin Johnson and Serena Yeung.

The starter code can be found at the course gdrive folder (you will need your andrew account to access):
https://drive.google.com/drive/folders/1JZGnpUG6Ca1o0q47PCxsQVQASQwJTmUu

We recommend editing and running your code in Google Colab, although you are welcome to use your local machine instead.

1 Pixel-based Nearest Neighbor Classifiacation

In this problem, we will implement the k-nearest neighbor algorithm to recognize objects in tiny images. We will use images from Imagenette [3], a small, easy-to-classify subset of ImageNet [2] (Figure 2). The code for loading and pre-processing the dataset has been provided for you.

Note: There is a `DEBUG` flag in the starter code that you can set to `True` while you are debugging your code. When the flag is set, only 20% of the training set will be loaded, so the rest of the code should take less time to run. However, before reporting the answers to questions, please remember to set the flag back to `False`, and to rerun the cells! There is also an option to run the code with a different image size, which you are welcome to experiment with (again, please set this back to the default before submitting!).

Figure 1: A sample of images from Imagenette [3], a small subset of ImageNet [2]

For the class `KNearestNeighbor` defined in the notebook, please finish implementing the following methods:

a. **(1 point)** Please read the header for the method `compute_distance_two_loops` and understand its inputs and outputs. Fill the remainder of the method as indicated in the notebook, to compute the L2 distance between the images in the test set and the images in the training set. The L2 distance is computed as the square root of the sum of the squared differences between the corresponding pixels of the two images.
Hint: You may use `np.linalg.norm` to compute the L2 distance.

b. **(1 point)** It will be important in subsequent problem sets to write fast *vectorized* code: that is, code that operates on multiple examples at once, using as few `for` loops as possible. Please complete the method. `compute_distance_one_loops` which computes the L2 distance only using a single for loop (and is thus partially vectorized).

c. **(1 point)** Please complete `compute_distance_no_loops` which computes the L2 distance without using any loops and is thus fully vectorized.
Hint: $||x - y||^2 = ||x||^2 + ||y||^2 - 2x^T y$

d. **(1 point)** Complete the implementation of `predict_labels` to find the k nearest neighbors for each test image.
Hint: It might be helpful to use the function `np.argsort`.

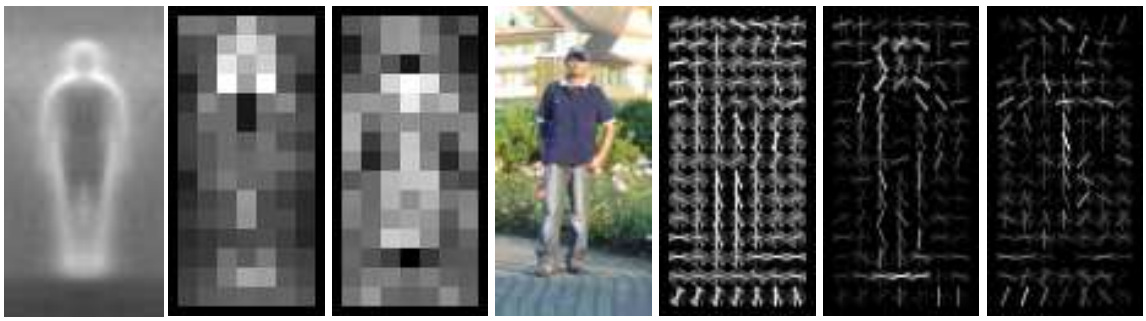
e. **(1 point)** Run the subsequent cells, so that we can check your implementation above. You will use `KNearestNeighbor` to predict the labels of test images and calculate the accuracy of these predictions. We have implemented the code for $k = 1$ and $k = 3$. For $k = 3$, you should expect to see approximately 25% test accuracy. Note that you may not get exactly the same accuracy as us because random shufflling may generate different train/test splits from us. Though this accuracy is somewhat low, this is far better than chance perfomance. Please answer the questions in the cell textbox.

f. **(1 point)** Implement normalization, where each image is shifted so that the average of all the (floating point and possibly negative) pixel values are zero. Also scale each image such that the sum of squared values of all the (floating point and possibly negative) pixel values are one.

g. **(1 point)** Prove that computing squared distance on normalized image descriptors is equivalent to computing the cosine distance between the two descriptors. We recommend using Markdown's latex functionality to do this in your notebook, but you may use another program to write out the proof and append it to the final PDF.

h. ***(0 point)*** Run the provided cells below to see the effects of normalization on the accuracy. You should be accuracy around 30%.

2 Histograms of oriented gradients (HOG) from scratch



(a) (b) (c) (d) (e) (f) (g)

Figure 6. Our HOG detectors cue mainly on silhouette contours (especially the head, shoulders and feet). The most active blocks are centred on the image background just *outside* the contour. (a) The average gradient image over the training examples. (b) Each "pixel" shows the maximum positive SVM weight in the block centred on the pixel. (c) Likewise for the negative SVM weights. (d) A test image. (e) It's computed R-HOG descriptor. (f,g) The R-HOG descriptor weighted by respectively the positive and the negative SVM weights.

Figure 2: Figure and caption from [1]. Note that we will not implement an SVM classifier.

Instead of finding the most similar images based on raw pixels, we obtain better performance using hand-crafted image features. We'll implement a Histogram of Oriented Gradients (HOG) descriptor [1], https://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf. You may wish to refer to the original paper. Note that scikit-image has its own implementation, but you may produce a *more* accurate one if you are careful! (it turns out that the scikit implementation makes some suboptimal choices at the image border). To compute this descriptor, you will:

a. **(1 point)** Compute the orientated gradients of an image by filling in the `compute_Gradient` function. Use modulo for angles that exceed 180 degrees so that all angles are in the range of [0, 180 deg]. We suggest first converting a color image to grayscale and then computing gradients. Once that is working, implement the approach (in the original HOG paper) that computes gradients separately for each color channel and for each pixel, selects the gradient with the largest norm.

b. (**1 point**) Create a histogram of edge orientations by filling the `bin_gradient` function. Weigh each edge's vote based on its gradient magnitudes. Each edge votes for *one* bin that its orientation falls in. You can make use of `math.floor()` to find the index of the bin.

c. **(1 point)** Perform block normalization across the histogram by filling in the `block_normalize` function.

Please read the descriptions in the starter code and fill in the code blocks. Please also run the cells below to test your code. Our debug implementation obtains about 35% accuracy.

3 Extra-credit: binning parameters (1 pt)

Because HOG descriptor is much smaller than the image itself, it can be computed on images larger than 32x32. Try larger images (e.g., 64x64) as input, which may require larger spatial bins in the histogram to keep the descriptor manageable in size. With simple parameter tweaks, we were able to increase accuracy over the default to 40%. Because these will require tweaking parameters in your notebook, we recommend adding a text cell to the notebook that reports the default-vs-modified parameters and the default-vs-final accuracy you obtained.

4 Extra-credit: low-rank descriptors (1 pt)

It turns out that distance calculations for nearest neighbors can be significantly sped up with low-rank approximations of descriptors. One can use principle component analysis (computed with an eigendecomposition or the singular value decomposition) to compute low-rank approximation of the original pixel image descriptors, their normalized variants, and the HOG descriptor. Low-rank approximations are very helpful when working with large images, since you may run into storage issues. Use the demo lecture code for eigenfaces to implement low-rank versions of the descriptors explored in this assignment. This will allow you to process bigger images, which can improve accuracy. In the case of HOG, you should be able to decrease the size of the descriptor by around 3X with little loss in accuracy. This is likely due to the fact that the overlapping block descriptors are redundant, which is an insufficiently well-known fact! We suggest adding an additional cell that computes low-rank versions of the original pixel image descriptors, their normalized variants, and the HOG descriptor. Report the reduction in descriptor size and the new accuracy.

References

[1] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. Ieee, 2005.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. 2009.

[3] J. Howard. Imagenette, 2020. URL https://github.com/fastai/imagenette.