EN.601.461/661 – Computer Vision
Fall 2017
Homework #3
Due: 11:59 PM, Sun, November 26, 2017

Changes

- 11/2 11:30pm fixes to last 2 convolution layers
- 11/8 3pm minor correction to last convolution layer (layer 13)
- 11/15 11:40am Added information on submitting the models.
- 11/20 2:20pm clarified you only need to submit models for which you used data augmentation.

All solutions (e.g., code, README write-ups, output images) should be zipped up as **HW3_yourJHED.zip** and posted on Blackboard, where 'yourJHED' is your JHED ID (ie.tkim60). Only basic Python library functions are allowed for part 2, unless otherwise specified. Part 1 is about deep learning and is thus quite different in terms of its limitations -- you should be able to use most opencv, pyplot, numpy and pytorch functions here. If you are unsure of an allowable function, please ask on Piazza before assuming! You will get a penalty for using a "magic" function to answer any questions where you should be answering them. When in doubt, ASK!

Submitting The Models

In addition to your code, you'll also have to submit your trained models. (Make sure to save your models as specified at the end of the pytorch introduction) This sounds simple until you realize that each model is 500MB, making the total 1GB **(only submit the models created with data augmentation)**!!! How do we do this? Well, first you'll pull the model files off of the VM using either SimpleHTTPServer or gcloud compute scp (see the gcloud document for details). Once they're on your computer, you'll upload the 2 model files to your google drive at drive.google.com. Place them in a directory called **HW3_yourJHED**. Once the 2 files are uploaded, you can share the files by right clicking on their directory (in Google Drive) and clicking on 'Get Shareable Link'. **Copy your link and paste it in your submission's README file on blackboard. Make sure not to modify your model files after submission time, or we'll have to deduct late days.** Also, keep those 2 model files around in your google drive until we're done grading homework 3, or we won't have access to them.

Programming Assignment

1) ***Make Deep Learning Happen With Faces:***

   Using the **Labeled Faces in the Wild** (LFW) dataset, along with the train/test splits that we provide to you, the goal of this exercise is to train variants of deep architectures (using your Google Cloud access) to learn when a pair of images of faces is the same person or not. Each of the following questions is designed to approach the problem in a slightly different manner, and then we ask you to analyze and say a few words about the performance of each in the end. For

the following, we will specify an exact architecture for you to implement (with PyTorch) and you will train and test a classifier to specify if a pair of images is the same or not. You will train only with the training split we give you, **with and without** data augmentation, and then report your average accuracy on the *train and test* split. Please read each part carefully!

You'll be using PyTorch for this assignment. PyTorch is a dynamic neural network library which was introduced in class. A good introduction to PyTorch can be found here, and the ipython notebook shown in class is here as a jupyter notebook or here copied to a google doc. You'll be using a combination of mostly high-level modules (torch.nn) and some lower-level code (torch) and utilities (torchvision) to create these networks.

To save your model in pytorch, use `torch.save(the_model.state_dict(), PATH)` to save, and `the_model.load_state_dict(torch.load(PATH))` to load. **Note that if you make the slightest change to your model, it'll no longer be compatible with the old saved weights!**

You'll also be using the google cloud (gcloud) for this assignment. The most important thing is for you to save your work, and not to leave the cloud VM running longer than you need to. **If you somehow exceed your assigned credits, report it to us immediately on Piazza!** More information about using gcloud can be found here. You can skip part 1 if you already followed the instructions on piazza.

As part of your submission, you must include programs **p1a.py** and **p1b.py**, and the saved model weights. **These programs must behave as follows**: they will assume the image data files are in the location ./lfw (ie. the directory lfw in their current directory), and that the test/train split files are located in the same directory. Passing the argument '--load WEIGHTS_FILE' to these programs (e.g. ./p1a.py --load my_weights1) will automatically load the saved network weights from the file WEIGHTS_FILE and test over both the train and test data, displaying accuracy statistics for both and anything else you want to show. Running the programs with '--save WEIGHTS_FILE' will cause the programs to train and save their weight data into WEIGHTS_FILE. What happens with any other arguments/no arguments is up to you -- just make sure you follow our interface for the arguments we want. Make sure when we run your program, we get the best parameters you've found.

[FOR ALL QUESTIONS, USE THE BUILT-IN ADAM OPTIMIZER WITH DEFAULT SETTINGS]

Consider a Convolutional Neural Network (CNN), denoted by **C**, that takes as input a single image *I* and outputs a feature vector $f \in \Re^N$, where *f* is simply the output of the final fully connected layer that contains *N* nodes (and hence, *N* numbers are produced). In other words, $C(I) = f \in \Re^N$. A **Siamese Network** is a CNN that takes two separate image inputs, I_1 and I_2, and both images go through the same exact CNN **C** (e.g., this is what's called "shared weights"), and so we can say: $C(I_1) = f_1$ and $C(I_2) = f_2$ (NOTE that we use the same **C** here, not two

different **C**'s!! Only one network -- this is key!). We can process the images in one pass, and then we can "play games" with the features that are output to learn what it means for two images (of faces) represented by their respective 'latent embeddings' (*f*) to be the same person/identity or not. Here's the base architecture we will use throughout. Let's call this **C**:

1. Convolution Layer
    Input: image of size 128x128x3 (3 for RGB color channels)
    Parameters: 64 features of size 5x5, stride of (1,1), padding=2
    Output: tensor of size 128x128x64
2. ReLU (in-place)
3. Batch Normalization Layer
    64 features
4. Max pooling layer
    Input: tensor of size 128x128x64
    Parameters: none, perform 2x2 max-pooling with (2,2) stride
    Output: tensor of size 64x64x64
5. Convolution Layer
    Input: 64x64x64
    Parameters: 128 features of size 5x5, stride of (1,1), padding=2
    Output: tensor of size 64x64x128
6. ReLU (in-place)
7. Batch Normalization Layer
    128 features
8. Max pooling layer
    Input: tensor of size 64x64x128
    Parameters: none, perform 2x2 max-pooling with (2,2) stride
    Output: tensor of size 32x32x128
9. Convolution Layer
    Input: 32x32x128
    Parameters: 256 features of size 3x3, stride of (1,1), padding=1
    Output: tensor of size 32x32x256
10. ReLU (in-place)
11. Batch Normalization Layer
    256 features
12. Max pooling layer
    Input: tensor of size 32x32x256
    Parameters: none, perform 2x2 max-pooling with (2,2) stride
    Output: tensor of size 16x16x256
13. Convolution Layer
    Input: 16x16x256
    Parameters: 512 features of size 3x3, stride of (1,1), padding=1
    Output: tensor of size 16x16x512
14. ReLU (in-place)
15. Batch Normalization Layer
    512 features

16. Flatten Layer
    Input: tensor of size 16x16x512
    Parameters: none, simply flatten the tensor into 1-D
    Output: vector of size 16x16x512=131072
    Note: this simple layer doesn't exist in pytorch. You'll have to use view(), or implement it yourself.

17. Fully Connected Layer (aka Linear Layer)
    Input: vector of size 131072
    Parameters: fully-connected layer with 1024 nodes
    Output: vector of size 1024
18. ReLU (in-place)
19. Batch Normalization Layer
    1024 features

For this network **C**, a feature *f* is produced as the output of layer number 19, of size 1024.

a. **Binary-classification with BCE**: We provided a train/test split (each) as a text file with rows containing image files paths of two images and then a 1 if they are the same person and 0 if not. You will read in these images and train the network as follows:

   Given the network topology described above, construct a siamese network that takes as input a pair of images into this shared network. Each image (from the pair) is run through **C** individually, producing f_1 and f_2. Combine f_1 and f_2 through concatenation to produce f_{12} (e.g., take the elements of f_1 and then combine the elements of f_2 to the end of f_1 so that f_{12} has dimensions 2*1024=2048). Then take the output of f_{12} and add one more fully-connected layer to classify this as 0 or 1. This output layer will have 1 node (e.g., only one number is output) and should have a sigmoid activation (so that the outputs are clipped in the range [0.0, 1.0]). The output of this last layer then represents the probability that this pair of images is the same (it'll be close to 1 if they are the same person and close to 0 if not). [Hint: train this with the binary cross-entropy loss]. To test, apply the trained network first to the train images and then to the test images and report the average classification error for both (note: you need to decide how to turn the sigmoid output of the last layer of the network into a discrete decision).

   You'll have to decide how many epochs (iterations over the data) to run for training. Report your decision in the README. *One way to know you're improving is to see your loss drop over time. Graph your loss after every session to see if it's dropping overall -- it's hard to know otherwise.*

   i. Do the same as above in a), but apply random data augmentation. This means, during the training loop, as images are read in, with some probability (use 0.7) apply a random transform to the images before running it through the network. The transform can consist of some (random) combination of: mirror flipping,

rotation (stay within -30 and +30 degrees, rotated with respect to the center of the image), translation (stay within -10 and +10 pixels) and scaling (stay within 0.7 to 1.3). Any borders that don't have valid image data make black. This should help to generalize better to the appearance of the faces and may have less chance of memorizing the training data. **Compare your average train and test accuracy with and without data augmentation in a separate PDF file with your submission**. [Note: the data augmentation only applies to the training, not the testing part]

To do data loading and augmentation, you may want to use the Transform, DataSet and DataLoader classes available in the torchvision package (which should be installed on google cloud).

ii. How well did part a work on the training data? On the test data? Any ideas why or why not it worked well or didn't? Answer this question below in part c.

b. **Contrastive-loss**: Now we will approach the problem in a slightly different way. Rather than attempt to directly classify the pair as 0 or 1, we will train the network to learn an embedding that attempts to bring features of same people closer to each other and separate apart people with different identities (e.g., the features representing all images of, say, "Brad Pitt" should be very close to each other, but far away from feature clusters of all other people). The way we will do this is to have the network learn a *similarity metric*. In other words, it's reasonable to assume just taking the L2-distance between features would achieve this (as in K-Means), and so we will train the network to optimize this objective to yield the best embedding that groups features via L2-distance. In the end, we will train with what's called *contrastive loss*, which is a custom loss function as follows. Suppose we define: $D_w(I_1, I_2) = \| C(I_1) - C(I_2) \|_2 = \|f_1 - f_2\|_2$ (e.g., the Euclidean distance between features produced by **C**, given the pair of input images). We then define our custom objective function *L* as:

$$L(I_1, I_2, y) = y * D_w(I_1, I_2)^2 + (1 - y) * \{max(0, m - D_w(I_1, I_2))\}^2$$

where I_1 and I_2 are the input image pairs, y is the ground truth label (1 if they are the same and 0 if they are different), and *m* is a margin term that nudges the network to separate features even more. You can start with a margin of 1.0, but feel free to use other margins and report your choice and reasoning in the README. The network is trained to minimize this term. Use the same base network as before. The only difference here is instead of concatenating features and doing binary classification, we are training to enforce structure on the data (think of it as supervised clustering). Use the same train/test split as before.

To test, we will do the same thing as part a: run your network over both the training data and the test data. This time, you should get a distance metric between the 2 images. You'll have to figure out a way to turn this distance metric into a prediction

decision whether the images are of the same person or not. Report accuracy metrics as before.

i. Do the same experiment but using data augmentation, as in a-i) above. In your report, compare performance of contrastive loss both with and without data augmentation.

ii. How well did contrastive loss work on the training data? How about the test data?

c. Please write a short paragraph (no more than 3-4 sentences) on comparing these two methods and why you think one worked better than the other, if that's the case, or why they performed the same, otherwise.

2) *Fundamental Matrix and Epipolar Lines [REQUIRED FOR GRADUATE STUDENTS ONLY]*

The purpose of this exercise is to estimate and display epipolar lines in the two images hopkins1.jpg (I1) and hopkins2.jpg (I2). Implement this in: ***fundamental.py***

a. Compute features in both images and match them (similar to what we did in assignment 2). Display the correspondences.

Aside: Though we would prefer to do SIFT here, it turns out to be more on the annoying side to get SIFT code to work in OpenCV (because it is patented). The goal here isn't to have you re-implement SIFT from scratch (that would be a homework assignment all on its own!). As I mentioned in class a few times, there are many alternatives to SIFT. This one called ORB is quite powerful and very fast (and conveniently enough, it's unpatented!). The high-level concept is the same--compute features and extract discriminable feature descriptors. We just want a good feature here for the next step. You may use the following function, freely available from scikit: http://scikit-image.org/docs/dev/api/skimage.feature.html#skimage.feature.ORB

Match features using Euclidean distances of feature descriptors as we mentioned in class and as you did in HW2. You may use the ratio test to get higher quality matches, and you will likely get some mismatches, as would be true with any feature method.

b. Find the fundamental matrix using the previously matched features. Use the method described in the lecture notes. *In order to get credit for this question, you may not simply call a vision library to compute the matrix for you. You must estimate it yourself!* You should use RANSAC to be robust to outliers.

c. Using the fundamental matrix computed above, draw the epipolar lines for a few selected features of your choice. (Randomly pick eight features in the left image and draw the eight corresponding epipolar lines on the right image).