CS 3430: SciComp with Py
Coding Exam 2

Vladimir Kulyukin
Department of Computer Science
Utah State University

March 20, 2018

Introduction

- This exam has two problems. You can code your solutions either in Py2 or Py3 (assuming you have OpenCV configured for Py3 on your computer or latop). If you code on the raspberry pi, OpenCV is already installed and configured.
- The folder `coding_exam_2/` in the zip archive contains the stub files `satyamitra_numbers.py`, `img_ann.py`, and `img_ann_data.py` you will code in and submit. If you code in Py2, save the stub files `coding_exam_2/p2/`, if in Py3, save them in `coding_exam_2/p3/`. Write your name and A-number at the beginning of each file before you submit your zip archive via Canvas.
- This exam is open books and open notes. If you use any online materials, please mention them in the comments at the beginning of the file where you used them.
- You may not collaborate with anyone on this exam orally, digitally, or in writing.
- You can code everything up on your laptop, desktop, or raspberry pi.
- You may use your homework solutions to solve the exam's problems. For problem 1, you must either write your own code to compute the determinant of a matrix or use your code from Assignment 6. You may not use the implementation of the determinant function from any third-party library. For problem 2, you may use your code from Assignment 7.
- Your solutions are due in Canvas by 11:59:59pm sharp today, Mar. 20, 2018. In other words, you have 9 hours to complete this exam.
- You are always better off submitting an incomplete solution for partial credit than a late complete solution for zero credit. To put it differently, don't email me your solutions after the Canvas submission closes. Canvas will close for submission at 11:59:59pm on Mar. 20, 2018.

Problem 1: 3 points

A *proper* factor of a whole number n is a whole number d that divides n with no remainder and is strictly less than n. For example, the proper factors of 8 are 1, 2, and 4. Implement the function `proper_factors`(n) that computes the list of all proper factors of an integer n. Here are a couple of examples.

```
>>> proper_factors(20)
[1, 2, 4, 5, 10]
>>> proper_factors(1000000)
[1, 2, 4, 5, 8, 10, 16, 20, 25, 32, 40, 50, 64, 80, 100, 125, 160, 200, 250, 320, 400, 500, 625,
800, 1000, 1250, 1600, 2000, 2500, 3125, 4000, 5000, 6250, 8000, 10000, 12500, 15625, 20000,
25000, 31250, 40000, 50000, 62500, 100000, 125000, 200000, 250000, 500000]
```

Two integers x and y are called *satyamitra* numbers if the sum of the proper factors of x is equal to y and the sum of the proper factors of y is equal to x. For example, 10856 and 10744 are satyamitra numbers. We can quickly check it as follows.

```
>>> sum(proper_factors(10744))
10856
>>> sum(proper_factors(10856))
10744
```

As an interesting historical note, the Pythagoreans, the followers of Pythagoras of Samos (c.570 - c.495 BC), an Ionian Greek philosopher, believed that if two friends wore amulets with two such numbers, they would fortify their friendship.

Implement the function `satyamitra_numbers_in_range(lower, upper)` that computes all pairs of satyamitra numbers in the range from `lower` to `upper` where both `lower` and `upper` are positive integers and `lower` $\leq$ `upper`. More specifically, a call to this function returns a list of pairs (i.e., 2-tuples) $[(x_1, x_2), (x_3, x_4), ..., (x_{k-1}, x_k)]$, where, in each pair (x_i, x_j), x_i and x_j are satyamitra numbers such that `lower` $\leq x_i \leq$ `upper` and `lower` $\leq x_j \leq$ `upper`.

Write the function `satyamitra_matrix(sn_list)` that takes a list of satyamitra number pairs that contains n numbers, where n is a perfect square. Recall that a perfect square is an integer that is the product of two equal integers, e.g., $4 = 2 \cdot 2$, $9 = 3 \cdot 3$, $16 = 4 \cdot 4$, etc. This function creates an $n \times n$ numpy matrix out of these pairs and computes its determinant.

For example, let `sn_list` $= [(x_1, x_2), (x_3, x_4)]$. Then `satyamitra_matrix(sn_list)` creates this matrix

$$\begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}$$

and then computes its determinant. Here is another example. Let

```
>>> sn_list = satyamitra_matrix(lower, upper)
```

Let's assume that `sn_list` contains 8 pairs of satyamitra numbers, i.e., a total of 16 numbers. In other words, `sn_list` $= [(x_1, x_2), ..., (x_{15}, x_{16})]$. Then `satyamitra_matrix` creates the following matrix

$$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ x_5 & x_6 & x_7 & x_8 \\ x_9 & x_{10} & x_{11} & x_{12} \\ x_{13} & x_{14} & x_{15} & x_{16} \end{bmatrix}$$

and computes its determinant.

Use this function to compute the determinant of the satyamitra numbers in [1, 20000]. There are 16 numbers in this range, i.e., 8 2-tuples. Save your solutions in `satyamitra_numbers.py`. In your comments to this problem, state the list of the satyamitra numbers and the value of the determinant.

Problem 2: 7 points

In this problem, you will build and train a 5-layer ANN to classify small black and white images. The zip archive contains two directories `img_black` and `img_white` that contains 200 images each. The directory `img_black` contains 200 5×5 black images and the directory `img_white` contains 200 5×5 white images.

Let's create the training data for the ANN. Implement the function `normalize_image(fp)` that takes the path to an image `fp`, reads it in, and converts it into a 1D normalized numpy array. Normalization in this case simply means dividing each pixel value by 255, the highest possible pixel value. Thus, each normalized array will have the values in $[0, 255]$. For example,

```
>>> normalize_image('img_black/0.png')
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
>>> normalize_image('img_white/10.png')
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

Write the function `create_data(img_dir, data_label)` that takes the image directory `img_dir` and `data_label` and goes through all image files in `img_dir` to create the list `DATA` of 3-tuples where the first element of each 3-tuple is an image path, the second element is the normalized array returned by calling `normalize_image` on the image path, and the third element is the `data_label`. For the black images, the `data_label` is equal to `np.array([0, 1])`; for the white images, the `data_label` is equal to `np.array([1, 0])`. Make sure that the list `DATA` is randomly shuffled with `np.random.shuffle`. Change the directories accordingly. Here's how you can create the training data.

```
>>> create_data('/home/pi/img_black/', np.array([0, 1]))
>>> create_data('/home/pi/img_white/', np.array([1, 0]))
>>> np.random.shuffle(DATA)
>>> DATA[0]
('/home/pi/img_white/120.png',
  array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
          1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]),
  array([1, 0]))
>>> DATA[1]
('/home/pi/img_black/120.png',
  array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
          0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]),
  array([0, 1]))
```

From the list DATA we need to create the numpy arrays X and y where X contains the training data that will be used to train your ANN and y contains the corresponding data labels, i.e., the ground truth. The function create_Xy(DATA) is provided for you in img_ann_data.py.

```
def create_Xy(DATA):
    global X, y
    for fp, img, yhat in DATA:
        X.append(img)
        y.append(yhat)
    X = np.array(X)
    y = np.array(y)
```

The elements of X are normalized arrays and the elements of y are the corresponding labels.

```
>>> X[0]
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
>>> y[0]
array([1, 0])
>>> X[11]
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
>>> y[11]
array([0, 1])
```

Now let's create the evaluation data with which you will evaluate your ANN after training it. Write the function create_eval_data(img_dir, data_label) that takes the image directory img_dir and data_label and goes through all images in img_dir to create the list EVAL_DATA of 3-tuples where the first element of each 3-tuple is an image path, the second element is the normalized array returned by calling normalize_image on the image path, and the third element is the data_label. For the black images, the data_label is equal to np.array([0, 1]); for the white images, the data_label is equal to np.array([1, 0]). Make sure that the list EVAL_DATA is randomly shuffled with np.random.shuffle. The zip archive contains two directories img_eval_black and img_eval_white that contain 200 images each that will be used in evaluating the trained ANN. Here's how you can create the evaluation data. Change the directories accordingly.

```
>>> create_eval_data('/home/pi/img_eval_black/', np.array([0, 1]))
>>> create_eval_data('/home/pi/img_eval_white/', np.array([1, 0]))
>>> np.random.shuffle(DATA)
>>> EVAL_DATA[0]
('img_eval_black/55.png',
  array([ 0.03921569,  0.03921569,  0.03921569,  0.03921569,  0.03921569,
        0.03921569,  0.03921569,  0.03921569,  0.03921569,  0.03921569,
        0.03921569,  0.03921569,  0.03921569,  0.03921569,  0.03921569,
        0.03921569,  0.03921569,  0.03921569,  0.03921569,  0.03921569,
        0.03921569,  0.03921569,  0.03921569,  0.03921569,  0.03921569]),
   array([0, 1]))
>>> EVAL_DATA[21]
('img_eval_white/134.png',
  array([ 0.98039216,  0.98039216,  0.98039216,  0.98039216,  0.98039216,
        0.98039216,  0.98039216,  0.98039216,  0.98039216,  0.98039216,
        0.98039216,  0.98039216,  0.98039216,  0.98039216,  0.98039216,
        0.98039216,  0.98039216,  0.98039216,  0.98039216,  0.98039216,
        0.98039216,  0.98039216,  0.98039216,  0.98039216,  0.98039216]),
  array([1, 0]))
```

From the `EVAL_DATA` list we can create the numpy arrays `EX` and `ey` where `EX` contains the training data that will be used to evaluate your ANN and `ey` contains the corresponding data labels, i.e., the ground truth. The function `create_EXey(EVAL_DATA)` is provided for you in `img_ann_data.py`.

```
def create_EXey(EVAL_DATA):
    global EX, ey
    for fp, img, yhat in EVAL_DATA:
        EX.append(img)
        ey.append(yhat)
    EX = np.array(EX)
    ey = np.array(ey)
```

The elements of `EX` are normalized arrays and the elements of `ey` are the corresponding labels.

```
>>> EX[0]
array([ 0.98039216,  0.98039216,  0.98039216,  0.98039216,  0.98039216,
        0.98039216,  0.98039216,  0.98039216,  0.98039216,  0.98039216,
        0.98039216,  0.98039216,  0.98039216,  0.98039216,  0.98039216,
        0.98039216,  0.98039216,  0.98039216,  0.98039216,  0.98039216,
        0.98039216,  0.98039216,  0.98039216,  0.98039216,  0.98039216])
>>> ey[0]
array([1, 0])
>>> EX[19]
array([ 0.03921569,  0.03921569,  0.03921569,  0.03921569,  0.03921569,
        0.03921569,  0.03921569,  0.03921569,  0.03921569,  0.03921569,
        0.03921569,  0.03921569,  0.03921569,  0.03921569,  0.03921569,
        0.03921569,  0.03921569,  0.03921569,  0.03921569,  0.03921569,
        0.03921569,  0.03921569,  0.03921569,  0.03921569,  0.03921569])
>>> ey[19]
array([0, 1])
```

Now implement the function `build_5_layer_nn(mat_dims)` in `img_ann.py` that will build weight matrices for a 5 layer ANN. It is completely up to you to decide on the dimensions of the layers. The only two obvious restrictions are that your ANN must have 25 neurons in the input layer and 2 neurons in the output layer. For example, if you call this function as `build_5_layer_nn((25, 100, 100, 50, 2))`, then this function returns a tuple of 4 weight matrices of small random float numbers created with `np.random.randn` where the first matrix is 25×100, the second matrix is 100×100, the third matrix 100×50, and the fourth matrix 50×2. By the way, these dimensions are just an example. I did not use them when I trained my ANN for this problem.

Implement the function `train_5_layer_nn(numIters, X, y, build)` that `numIters` is the number of iterations that your 5-layer ANN will be trained, `X` is the training data created in `img_ann_data.py`, `y` is the ground truth for the training data also created in `img_ann_data.py`, and `build` is the function that builds your 5-layer ANN network.

The function `fit_5_layer_nn(x, wmats, thresh=0.4, thresh_flag=False)` that fits the input `x` and a 4-tuple of the weight matrices `wmats` of a trained 5-layer ANN, and classifies the input as `np.array([1, 0])` or `np.array([0, 1])` when `thresh_flag=True` and as two-element numpy arrays of floats if `thresh_flag=False`.

Use the the following function `eval_img_nn(fit_fun, wmats, Ex, ey)` is implemented in `img_ann.py` to evaluate your ANN on the evaluation data.

```
def eval_img_nn(fit_fun, wmats, EX, ey):
    count = 0
    acc = 0
    for i in xrange(len(EX)):
        if np.array_equal(fit_fun(EX[i], wmats, thresh_flag=True),
                          ey[i]) == True:
            acc += 1
        count += 1
    return float(acc)/count
```

This function takes the ANN fit function `fit_fun` such as `fit_5_layer_nn`, the weight matrices of the trained ANN `wmats`, the evaluation data `EX`, and the ground truth of the trained data `ey` and computes the evaluation accuracy. The file `img_ann.py` also contains `find_best_nn` that you can use to find your best trained ANN.

```
def find_best_nn(lower_num_iters, upper_num_iters, step, train_fun,
                 fit_fun, eval_fun, bn, X, y):
    for numIters in xrange(lower_num_iters, upper_num_iters, step):
        wmats = train_fun(numIters, X, y, bn)
        acc = eval_fun(fit_fun, wmats, X, y)
        print numIters, acc
        if acc > 0.8:
            return wmats
    return None
```

This function takes the lower and upper bounds for the number of iterations for training your ANN. The interval I used in my training is `[1000, 200000]`. The `step` is the increment step value that we use to move from the lower bound to the upper bound. I used a step value of `1000`. The `train_fn` is an ANN training function such as `trian_5_layer_nn`, the `fit_fun` is an ANN fitting function such as `fit_5_layer_nn`, the `eval_fun` is an ANN evaluating function such as `eval_fun`, the `bn` is an ANN building function, and the `X`, and the `y` are the trained data and ground truth, respectively. Here is the call I used to train my ANN. The function `bn1` is a function that builds an initial 5-layer ANN that is being trained.

```
>>> wmats = find_best_nn(1000, 200000, 1000, train_5_layer_nn,
                 fit_5_layer_nn, eval_img_nn, bn1, X, y)
1000 0.0
2000 0.0
3000 0.0
4000 0.0
5000 0.0
6000 0.0
7000 0.5
8000 0.0
9000 0.0
10000 0.5
11000 0.5
...
87000 1.0
```

This function call returns as soon as the network's accuracy is above 80%. As the above output shows, I managed to train an ANN with an accuracy of 100% after 87000 iterations. Another ANN that achieved this accuracy took considerably more iterations. You number of iterations will most likely be different, because the initial weights are randomly initialized.

In `img_ann.py`, write two functions `pickle_nn(fp, wmats)` and `upickle_nn(fp)`. The first one takes the array of the weight matrices of the trained ANN returned by `find_best_nn` and pickles it in the file specified by the file path `fp`. The function `unpickle_nn` takes the file path `fp` where the pickled ANN weight matrices are saved and unpickles it. Here's how you can pickle, unpickle, and evaluate the unpickled ANN.

```
>>> pickle_nn('/home/pi/exam02_ann.pck', wmats)
>>> ann_wmats = unpickle_nn('/home/pi/exam02_ann.pck')
>>> eval_img_nn(fit_5_layer_nn, wmats, EX, ey)
1.0
```

What to Submit

If you code in Py2, zip and submit the following files on Canvas by 11:59:59pm sharp on Mar. 20, 2018.

1. `py2/satyamtra_numbers.py`,
2. `py2/img_ann.py`,
3. `py2/img_ann_data.py`.

If you code in Py3, zip submit the following files on Canvas by 11:59:59pm sharp on Mar. 20, 2018.

1. `py3/satyamtra_numbers.py`,
2. `py3/img_ann.py`,

3. `py3/img_ann_data.py`.

Your zip archive should also contain the pickled file of your best ANN's weight matrices saved as `exam02_ann.pck`. Do not include the images in your zip.

Happy Hacking!