Task 1: Flatten and unflatten images

The images we are working with are represented by a 2D list (i.e., a list of lists of zeros and ones) of 28x28 dimensions. In order to work with these lists, we need to be able to i) flatten the 2D representation and ii) unflatten the flattened representation.

For example, consider a smaller 5x5 2D list (i.e., image) that depicts a cross:

image = [

[1, 0, 0, 0, 1],

[0, 1, 0, 1, 0],

[0, 0, 1, 0, 0],

[0, 1, 0, 1, 0],

[1, 0, 0, 0, 1]

]

The flattened version of this list (i.e., flattened\_image) would be:

flattened\_image = [1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1]

The function flatten\_image(image) has been provided for you as part of the assignment scaffold.

Task 1:

Create a function unflatten\_image(flat\_image) that takes in a 1D list of any size (assuming it is a valid flattened representation) and returns the 2D representation.

**Hint:** Given you are not allowed to import and use the math module, you can write x\*\*0.5 instead of math.sqrt(x) in order to take the square root of x.

Please note that you must not assume the size of input images, only that they will be *n* x *n* where *n* is a positive integer.

I have completed this task. The code is attached.

Task 2: Checking for an adjacent 1

The generation of new image(s) from an input image of the same number will be done by flipping pixels from 0s to 1s under certain constraints. One of those constraints is that a pixel can only be flipped from 0 to 1, and such a flip is only allowed if there is another adjacent pixel in the input image with the value of 1. In order to do this, we need a function that checks if a pixel has an adjacent pixel with the value of 1 in the input image. Here, adjacency is defined as the four orthogonal directions (i.e., up, down, left, right) which does not include the diagonals.

Consider the example image below that creates an "L" shape, with a focus on two pixels of the value 0 that have been highlighted in bold:

image = [

[1, 0, 0, 0, 0],

[1, 0, 0, **0**, 0],

[1, 0, 0, 0, 0],

[1, 0, **0**, 0, 0],

[1, 1, 1, 1, 1]

]

The pixel with the indices (1,3) of image does not have any pixels with the value of 1 adjacent to it whereas the pixel with the indices (3,2) does.

Note that in the flattened representation of image (i.e., flat\_image), the two pixels with indices (1,3) and (3,2) correspond to the indices 8 and 17, respectively:

flat\_image = [1, 0, 0, 0, 0, 1, 0, 0, **0**, 0, 1, 0, 0, 0, 0, 1, 0, **0**, 0, 0, 1, 1, 1, 1, 1]

Task 2:

Create a function check\_adjacent\_for\_one(flat\_image, flat\_pixel) that checks if a given pixel in a flattened image has an adjacent pixel with the value of 1.

Remember to make use of your previously written functions flatten\_image and/or unflatten\_image to achieve this task.

Please note that you must not assume the size of input lists, only that they will be *n* x *n* where *n* is a positive integer.

Have completed this task as well. PFA the file.

Task 3: Recursively flipping pixels

In order to generate new image(s), we need a function that can generate all possible new unique images from the input image by the means of flipping pixels from 0s to 1s under the following two constraints:

1. a pixel can only be flipped from 0 to 1, and such a flip is only allowed if there is another adjacent pixel in the input image with the value of 1, and
2. the number of pixels that can be flipped is limited by a budget.

For example, consider the 3x3 example image below depicting an "L" shape:

image = [

[1, 0, 0],

[1, 0, 0],

[1, 1, 1]

]

One possible image that satisfies with our adjacency constraint is:

[1, 0, 0]

[1, 0, **1**]

[1, 1, 1]

Only one pixel is flipped in this example. Note that even if our budget was 2, as long as the number of pixels flipped is less than or equal to the budget, it is considered a valid possibility.

The below possible image contains two flips:

[1, 0, 0]

[1, **1**, **1**]

[1, 1, 1]

Note that in this example, the top right pixel should never be flipped as it was not adjacent to any pixel of value 1 in the original input image.

Finally, please note that you **must** use recursion to accomplish this task for full marks. If you do not use recursion (i.e., implement a solution that is not based on recursion), you can achieve up to half of the marks.

Task 3:

Create a function pixel\_flip(lst, orig\_lst, budget, results, i=0) that uses recursion to generate all possible new unique images from the input orig\_lst, following these rules:

* The input lst is the current list being processed. Initially, this will be the same as orig\_lst which is the original flattened image.
* The input budget represents the number of pixels that can still be flipped. When the budget reaches 0, no more pixels can be flipped.
* The input results is a list of resulting flattened images with flipped pixels. Initially, this will be an empty list.
* The input i represents the index of the pixel being processed, by default set to 0, which is used to drive the recursive function towards its base case (i.e., initially starting from i=0).

At termination of the function, the argument results should contain all possibilities of the input orig\_lst by only flipping pixels from 0 to 1 under both the budget and the adjacency constraints.

Please note that you must not assume the size of input images, only that they will be *n* x *n* where *n* is a positive integer.

Note : Automated tests for this task will take some time to execute as they will also test your solution on a 28x28 image. Please simply re-run the marker if you receive the message 'Your program ran for too long'.

You can use read\_image from the ai module provided along with the example images to try pixel\_flip on a full size 28x28 image.

PFA the template

## Task 4: Writing images to a file

Now that we have the capability of generating new images, we can begin working on storing the generated images as text files. In order to do so, we need the capability of writing the newly generated images to a file with a marker indicating the flipped pixel(s).

Reading images can be done using the function read\_image within the provided ai module.

## Task 4:

Create a function write\_image(orig\_image, new\_image, file\_name) that takes a 2D list of integers representing the original image (i.e., orig\_image), a 2D list of integers representing a newly generated image (i.e., new\_image), and a string representing the filename for the output file (i.e., file\_name). The function should write the new image into the output file, marking the modified pixels with 'X' to indicate the difference(s) from the original image.

Please note that you must not assume the size of input images, only that they will be *n* x *n* where *n* is a positive integer.

PFA the template

## Task 5: Generating new images

In this task, you will make use of the functions you have previously created in order to define a new function which will generate all possible new unique images from the input image with the same predicted number (i.e., based on the prediction of the predict\_number function) that satisfy the previously specified budget and adjacency constraints.

The provided ai.py module contains the predict\_number function that accepts as an argument a 2D list of integers representing an image and returns an integer for the number it predicts the image represents.

## Task 5:

Create a function generate\_new\_images(image, budget) that takes a 2D list of integers representing an image (i.e., image), and an integer representing the number of pixels that can be flipped (i.e., budget).

The function should input a 2D list and generate all possible new unique images from the input image under the previously specified budget and adjacency constraints (i.e., using the previously implemented pixel\_flip and check\_adjacent\_for\_one functions) while ensuring that the number predicted in the generated images are the same as that of the original input image (i.e., based on the prediction of the predict\_number function).

The function should return the final list of generated images.

Remember to use predict\_number from ai to test the prediction for each generated image.

Remember to flatten/unflatten images when needed.

Please note that this task does not have any automated test cases, you will get a chance to test your solution during the final task.

## Task 6: Testing generated images

For the final task, we are going to create a unit test to verify the correctness of both our solutions and the generated images, and ensure that they comply with our adjacency and budget constraints. We will do this by implementing each test case as a method in our unit test in the file named test\_generative.py.

## Task 6:

Using assertions and the unittest module, please create a test case for each of the testing criteria that is presented below:

* Verify output of flatten\_image for at least three different sizes of images.
* Verify output of unflatten\_image for at least three different sizes of flattened images.
* Verify output of check\_adjacent\_for\_one for three different pixel indexes of an image representing different scenarios (such as in the middle of an image, on the edge and in a corner).
* Verify output of pixel\_flip for a 5x5 image with a budget of 2. Ensure that the image has at least 8 possible locations where a 0 can be flipped to a 1.
* Verify output of generate\_new\_images with the input image.txt such that each output image satisfies:
  + image is of size 28x28,
  + all values in the generated image are either 0s or 1s,
  + the number of pixels flipped from the original input image is within the specified budget,
  + all pixels flipped from the original image had an adjacent value of 1 in the input image.

Each test case's name must begin with test\_ in order for it to be executed under unittest.

Given the purpose of this task is to test the correctness of your code using unit testing, there are no automarkers for this task. Please note that the unit test can take a long time (e.g., 90 seconds or more).