Handin 5

Due 11 Oct by 18:00 **Points** 100 **Submitting** an external tool
Available after 4 Oct at 18:00

Part 1

This week's handin will be about numpy and matplotlib. Let's start with a small numpy warmup exercise:

1. In the first weeks of the course, when we talked about looping, one of the exercises was to loop through this list, `[6,9,4,8,7,1,2,1]`, and figure out which elements which were larger than the left neighbor (i.e. 9, 8, and 2). The task today is to do a similar exercise using numpy, WITHOUT using loops. Inside `handin5.py` create a function called `count_larger_than_neighbor_without_loops` that takes a single argument, `list_of_numbers` as argument. The function should convert the input list to a numpy array. Then, using numpy operations, without writing a loop, it should determine how many entries in this array are larger than their left-neighbor, and return this result. Hint: you can solve this by creating two sub-arrays (slices) of the original array, and comparing them.

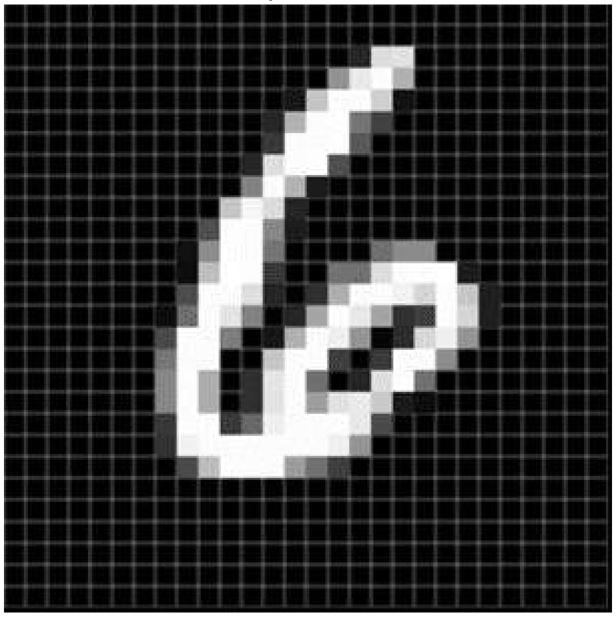
   Inside `handin5_test.py`, call the `count_larger_than_neighbor_without_loops` function on `[6,9,4,8,7,1,2,1]` and save the result in a variable called `larger_than_neighbor_count`.

We will now get some more experience with numpy and matplotlib by working on a classic Machine Learning data set called MNIST. MNIST is a data set of handwritten digits (see **https://en.wikipedia.org/wiki/MNIST_database (https://en.wikipedia.org/wiki/MNIST_database)** for details). The full dataset has 70,000 images - but for convenience I have reduced it to only 200 images, and made it available for download here: **https://wouterboomsma.github.io/ppds2023/data/mnist_test_200.csv (https://wouterboomsma.github.io/ppds2023/data/mnist_test_200.csv)**.

2. In the module file called `handin5.py`, create a function called `read_mnist_csv`, which takes a single argument called `filename`. This function should read the file, and return a numpy array. Note that the first line in the .csv file is a header with column names, which you should exclude when reading in the data. Hint: you can use `genfromtxt` for this task.

   Verify that your function works by calling it from the `handin5_test.py` file, on the filename `mnist_test_200.csv`. Save the result in a variable `mnist_data`.

3. Each row in the csv file consists of 785 values: 1 label - a numbers between 0 and 9 that specifies which digit it corresponds to, and 784 pixel values, corresponding to an 28x28 image (the images are in grey-scale, and a pixel value value just specifies how white a pixel is: 0 is black, 255 is white). Here is an example:

   Inside `handin5.py`, write a function called `group_by_label` that takes a numpy array as argument (similar to the one we created in the previous question), and returns a list of numpy arrays, grouped by which digit it corresponds to. The output should thus be a list with 10 elements and, for example, the first element in this list is a numpy array corresponding to all the rows in the original array where the first column is 0 (i.e., the first of the 10 digits). Note that we keep all 785 columns in these numpy arrays (including the digit label).

   In the `handin5_test.py` file, verify that your code works by calling the `group_by_label` function, using the numpy array from the previous exercise as argument. Save the result in a variable called `digit_image_groups`.

4. Inside `handin5.py`, write a function called `get_image` that takes two arguments: 1) a list of numpy arrays called `digit_image_groups` (the one created by the previous exerise), and 2) an argument called `digit`. The function should use the `digit` argument to select the relevant group in `digit_image_groups`, and then in this group choose a random image, which should then be returned to the user as a 28x28 numpy array (i.e. omitting the very first column which specifies the digit identity). Hint: you can use the `reshape` method in any numpy array to change the shape of an array (while keeping the total number of elements fixed).

   In the `handin5_test.py` file, call the `get_image` function with some digit value, and save the result in a variable called `digit_image`. Use matplotlib's `imshow` function to visualize your images, and save it to a file called `random_image.png`. Set the color map (cmap) to gray and remove the axis on the plot.

Part 2: Project

The last weeks, we have processed temperature anomaly data from the `Land_and_Ocean_summary.txt` file in various ways. We now want to visualize the data. Some years ago, Ed Hawkins from the University of Reading created a simple, but striking visualization of this type of data: **https://showyourstripes.info/ (https://showyourstripes.info/)**. His visualization has since been used on neckties, necklaces, coffee mugs etc - and even made the cover of the economist (**https://www.economist.com/leaders/2019/09/19/the-climate-issue (https://www.economist.com/leaders/2019/09/19/the-climate-issue)**). We will try to reproduce this plot.

1. We'll start by parsing the data once again, this time using Numpy. Inside `handin5_project_test.py`, use the `genfromtxt` function from numpy to read the `Land_and_Ocean_summary.txt` data into a numpy array. You will need to specify how you want `genfromtxt` to deal with the comment lines. Save the result in a variable called `anomaly_data`. Verify that you can extract the year values and the anomaly values from this numpy array. For the remainder of this exercise, when I write "anomaly" I mean the "Land + Ocean anomaly using air temperature above sea ice", which is the second column (the first after the year).

2. To reproduce the visualization, we need to somehow convert temperature anomaly values to colors. We will do this by writing a class called `ColorMapper`. The idea is that whenever we create a `ColorMapper` object, we specify a minimum and maximum temperature anomaly value. Afterwards, we can then use this object to get color values corresponding to a particular temperature anomaly value. Luckily for us, most of this functionality already exists in matplotlib. Matplotlib color maps (cmaps) are functions that take a number between 0 and 1 as input, and return a color as output. This color will be a tuple of four values: (red, green, blue, alpha), where the first three specify the degree of red, green and blue in the color, and alpha specifies the opacity of the color. For instance, if we set `cmap = plt.get_cmap("RdBu")`, `cmap(0.5)` will produce (0.97, 0.97, 0.97, 1.0), which is (almost) white, cmap(0.0) returns (0.40, 0.0, 0.12, 1.0), which is a redish color, and cmap(1.0) produces (0.02, 0.19, 0.38, 1.0), which is a blueish color (`get_cmap("RdBu_r")` will do the opposite, starting in blue and ending in red). All we need to do is therefore to convert our anomaly values to be between zero and one. Here is a draft of such a class.

```
class ColorMapper:
    """
    Wrapper class for matplotlib cmap, that maps a range of values to
    a [0,1] range, such that values can be mapped to colors.
    """

    def __init__(self, values, cmap_str='RdBu_r'):
        """
        Constructor. Extracts min and max values from a numpy array of values, and uses
        this to determine how to map values to colors.

        :param values: Numpy array of float values
        :param cmap_str: String argument for color map, which will be sent to matplotlib.
        """

        # Save attributes
        self.cmap = plt.get_cmap(cmap_str)
        self.min_value = np.min(values)
        self.max_value = np.max(values)

    def get_color(self, value):
        """
        Retrieve color corresponding to a particular value.
        """
        return self.cmap((value - self.min_value) / (self.max_value - self.min_value))
```

Copy this class to your `handin5_project.py` file. In your `handin5_project_test.py` file instantiate a new `ColorMapper` object using the anomaly value column from the `anomaly_data` array. Save this object in a variable called `color_mapper`. Verify that you can call the `get_color()` method to translate anomaly values to colors.

There is something wrong with our implementation of `ColorMapper`. If we input an anomaly of zero, we would like that to correspond to the color white. Change the min_value and/or max_value attributes in the class to make this work, by ensuring the the [min_val; max_val] range is centered at zero. Note that even after fixing it, the red, green, blue color will not be exactly 1.0 - but it should be relatively close (around 0.96). Hint: solve this by figuring out the maximum absolute value of the two temperatures (i.e. which of positive and negative values is furthest from zero), and then set the min to the negative of this number and the max to the positive of this number.

3. Finally, we need to create the stripes for the visualization. For each year in our data set, it should creates a rectangle with a particular color depending on the anomaly value. Inside `handin5_project.py`, write a function called `construct_rectangles` that takes 4 arguments: 1) `years`, which is a sequence of year values, 2) `anomalies`, which is a sequence of anomaly values, 3) `bottom`: a value specifying the lower coordinate value of each stripe, which should default to 0.0, and 4) `height`: a value specifying the height of each stripe, which should default to 1.0. The function should return a list of tuples, each with 5 elements: (year, bottom, width, height, temperature anomaly), where width is the width of the stripe, measured in years, which we can just set to 1.

   Inside the `handin5_project_test.py` file, call the construct_rectangles function and save the result in a variable called `rectangles`. To verify that it worked, call the following provided function, which should produce a visualization close to the original one by Ed Hawkins.

```
def plot_stripes(list_of_rectangles,
                 color_mapper,
                 colorbar=True,
                 figure_width=20, figure_height=5):
    """
    Visualize list of rectangles, where each rectangle is specified in the format
    (x-coordinate, y-coordinate, width, height, value). The color_mapper is
    used to look up colors corresponding to the values provided for each rectangle.

    :param list_of_rectangles: List of (x-coordinate, y-coordinate, width, height, value) tuples
    :param color_mapper: ColorMapper object used to lookup values for each block
    :param colorbar: Whether to include a color bar
    :param figure_width: Width of figure
    :param figure_height: Height of figure
    :return: None
    """

    _, ax = plt.subplots(1, figsize=(figure_width, figure_height))
    x_values = []
    y_values = []
    for rectangle in list_of_rectangles:

        anomaly_value = rectangle[-1]

        rect = matplotlib.patches.Rectangle(rectangle[:2], rectangle[2], rectangle[3],
                                            linewidth=1, edgecolor='none',
                                            facecolor=color_mapper.get_color(anomaly_value))
        ax.add_patch(rect)
        x_values += [rectangle[0], rectangle[0]+rectangle[2]]
        y_values += [rectangle[1], rectangle[1]+rectangle[3]]

    ax.set_xlim(min(x_values), max(x_values))
    ax.set_ylim(min(y_values), max(y_values))

    if colorbar:
        from mpl_toolkits.axes_grid1 import make_axes_locatable
        divider = make_axes_locatable(plt.gca())
        ax_cb = divider.new_horizontal(size="1%", pad=0.1)
        matplotlib.colorbar.ColorbarBase(ax_cb, cmap=color_mapper.cmap,
                                         orientation='vertical',
                                         norm=matplotlib.colors.Normalize(
                                             vmin=color_mapper.min_value,
                                             vmax=color_mapper.max_value))
        plt.gcf().add_axes(ax_cb)
    plt.show()
```

This tool needs to be loaded in a new browser window

Load Handin 5 in a new window