Data Set Preparation

Solve the following exercises and upload your solutions to Moodle (unless specified otherwise) until the specified due date. Make sure to use the *exact filenames* that are specified for each individual exercise. Unless explicitly stated otherwise, you can assume correct user input and correct arguments. You are allowed to write additional functions, classes, etc. to improve readability and code quality.

Exercise 1 – Submission: `a2_ex1.py` 40 Points

Write a function `to_grayscale(pil_image: np.ndarray) -> np.ndarray` that converts a `pil_image` (expected to be the raw data of an image loaded with Pillow) to grayscale using the colorimetric conversion. The conversion works on an RGB input image where all values must have been normalized to the range $[0, 1]$ before the (also normalized) grayscale output Y is calculated as follows:

$$C_{\text{linear}} = \begin{cases} \frac{C}{12.92} & \text{if } C \leq 0.04045, \\ \left(\frac{C+0.055}{1.055}\right)^{2.4} & \text{otherwise.} \end{cases} \qquad \text{with } C \in \{R, G, B\}$$

$$Y_{\text{linear}} = 0.2126 \cdot R_{\text{linear}} + 0.7152 \cdot G_{\text{linear}} + 0.0722 \cdot B_{\text{linear}}$$

$$Y = \begin{cases} 12.92 \cdot Y_{\text{linear}} & \text{if } Y_{\text{linear}} \leq 0.0031308, \\ 1.055 \cdot Y_{\text{linear}}^{\frac{1}{2.4}} - 0.055 & \text{otherwise.} \end{cases}$$

The function must return the grayscale-converted image *including* a dedicated channel for the brightness information in the following 3D shape: `(1, H, W)`, where `H` is the height and `W` the width of the image. The specified `pil_image` must be handled as follows:

- If the image has a 2D shape, it is assumed to already be a grayscale image with shape `(H, W)`, and a copy with shape `(1, H, W)` is returned.
- If the image has a 3D shape, it is assumed to be `(H, W, 3)`, i.e., the third dimension represents the RGB channels (in the order R, G and B, which can also be assumed to be true). If the third dimension does not have a size of exactly three, a `ValueError` must be raised.
- If the image has a different shape, a `ValueError` must be raised.
- You can assume that the images passed to the function will only have values within the range $[0, 255]$. This enables an easy normalization by dividing all image values by 255. However, the original input image must not be changed, i.e., the normalization and grayscale conversion must be done on a copy. Do not forget about the denormalization of Y before you return the grayscale-converted image.
- The data type of the image can be arbitrary. If it is an `np.integer` (use `np.issubdtype`), then rounding (to zero decimals) must be applied before returning the grayscale-converted image. The returned grayscale-converted image must have the same data type as the input image, so make sure to cast appropriately.

Hints:

- The function `np.where` might be useful when implementing the mathematical formulae above.
- Avoid manual loops, utilize broadcasting instead.
- Do not forget about the additional brightness channel before returning the result (both when the input is already a grayscale image and when it is an RGB image).

Exercise 2 – Submission: `a2_ex2.py` 60 Points

Write a function

```
prepare_image(
    image: np.ndarray,
    x: int,
    y: int,
    width: int,
    height: int,
    size: int
) -> tuple[np.ndarray, np.ndarray, np.ndarray]
```

that prepares an `image` for the depixelation machine learning project. Its main task is to create the pixelated image by pixelating a rectangular area that is specified via the parameters `x`, `y`, `width`, `height` and `size`. All parameters are explained in the following:

- `image` is a 3D NumPy array with shape `(1, H, W)` that contains a grayscale image. This image must be pixelated according to the area specified by the remaining parameters (cf. Figure 1).
- `x` specifies the x-coordinate within `image` where the pixelated area should start (cf. Figure 1).
- `y` specifies the y-coordinate within `image` where the pixelated area should start (cf. Figure 1).
- `width` specifies the width of the pixelated area (cf. Figure 1).
- `height` specifies the height of the pixelated area (cf. Figure 1).
- `size` specifies the block size of the pixelation process (cf. Figure 2).

The pixelation process works as follows: Within the area that should be pixelated, a square block of `size` × `size` is used to calculate the average pixel value, and then, all pixels within this block are replaced by this average pixel value.[1] Afterwards, the block is moved to the next non-overlapping position in the pixelated area (i.e., the step size of the block is also `size` in both x- and y-directions), and the above procedure (calculate average and overwrite) is repeated. This is done until the end of the pixelated area is reached. If `width` and/or `height` are not evenly divisible by `size`, the last blocks on the right and/or bottom will not properly fit the pixelated area. In such cases, the block size is shrunk accordingly. See Figure 2 and Figure 3 for an example visualization.

[1] If the input image has an integer data type, then the average is simply converted to an integer (i.e., no rounding is applied) when overwriting pixel values.

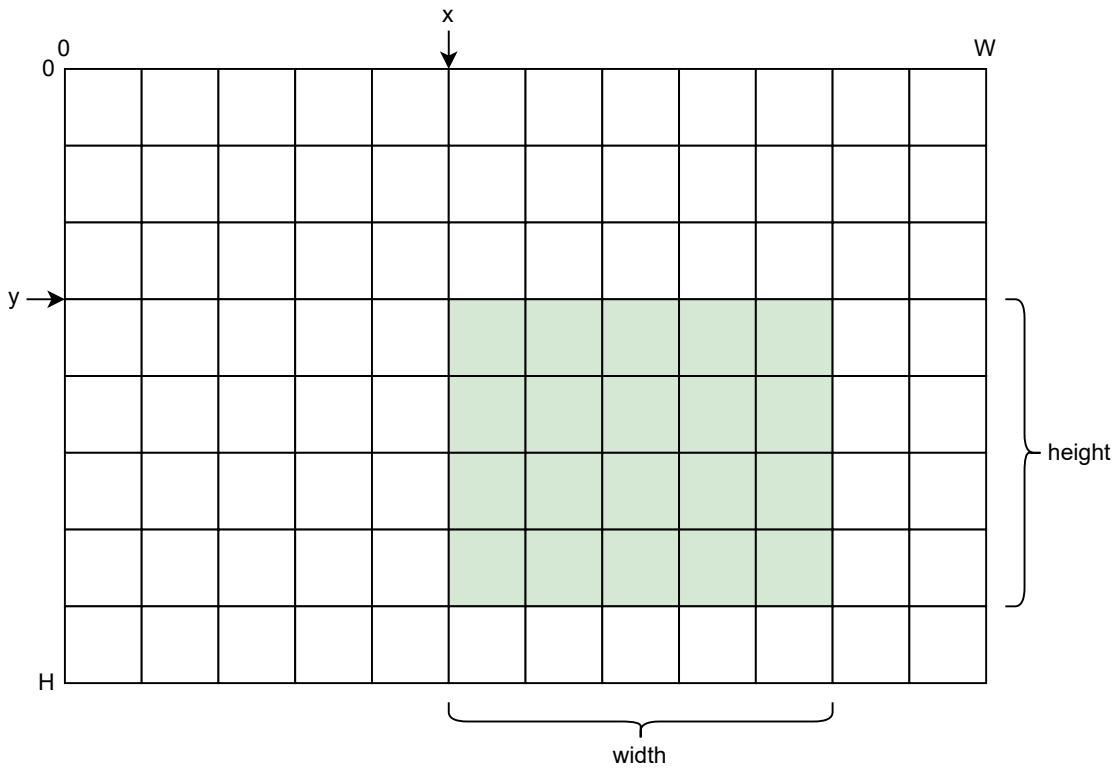

Figure 1: Example visualization of the pixelated area (green) within the input image of shape `(H, W)`, given some user specified arguments for `x`, `y`, `width` and `height`.

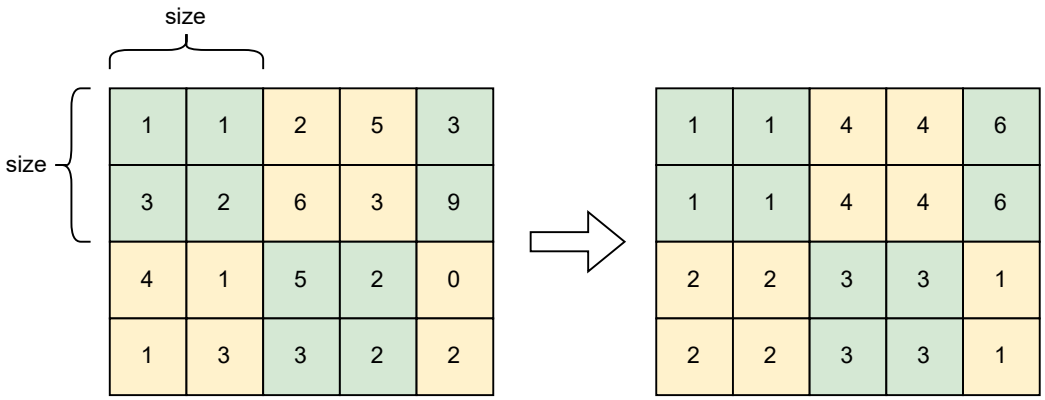

Figure 2: Example of how the block size of the pixelation process works (within the pixelated area). In this example, the width of the pixelated area (5) is not evenly divisible by `size` (2), so the last blocks on the right are shrunk (to size $5 \% 2 = 1$). For each block, its average pixel value is calculated, and then, all its pixels are replaced by this average.

The function must return the 3-tuple `(pixelated_image, known_array, target_array)`:

- `pixelated_image` is the 3D NumPy array that represents the pixelated version of the input image (cf. Figure 3). The array must be a copy, i.e., the original input image must not be changed. It must have the same data type and shape `(1, H, W)` as the input image.
- `known_array` is a boolean 3D NumPy array that has entries `True` for all original, unchanged pixels in `pixelated_image` and `False` for all unknown, pixelated pixels in `pixelated_image` (cf. Figure 3). It must have the same shape `(1, H, W)` as `pixelated_image`.
- `target_array` is a 3D NumPy array that represents the original pixels of the pixelated area before the pixelation process (cf. Figure 3), i.e., all pixels of the input image where `known_array` is `False`. It must have the same data type as the input image.

The function must also handle various error cases. Specifically, it must raise a `ValueError` in the following cases:

- The shape of `image` is not 3D or the channel size is not exactly 1 (i.e., shape `(1, H, W)`).
- Either `width`, `height` or `size` are smaller than 2.
- `x` is smaller than 0 or `x + width` is larger than the width of the input image, i.e., the pixelated area would exceed the input image width.
- `y` is smaller than 0 or `y + height` is larger than the height of the input image, i.e., the pixelated area would exceed the input image height.

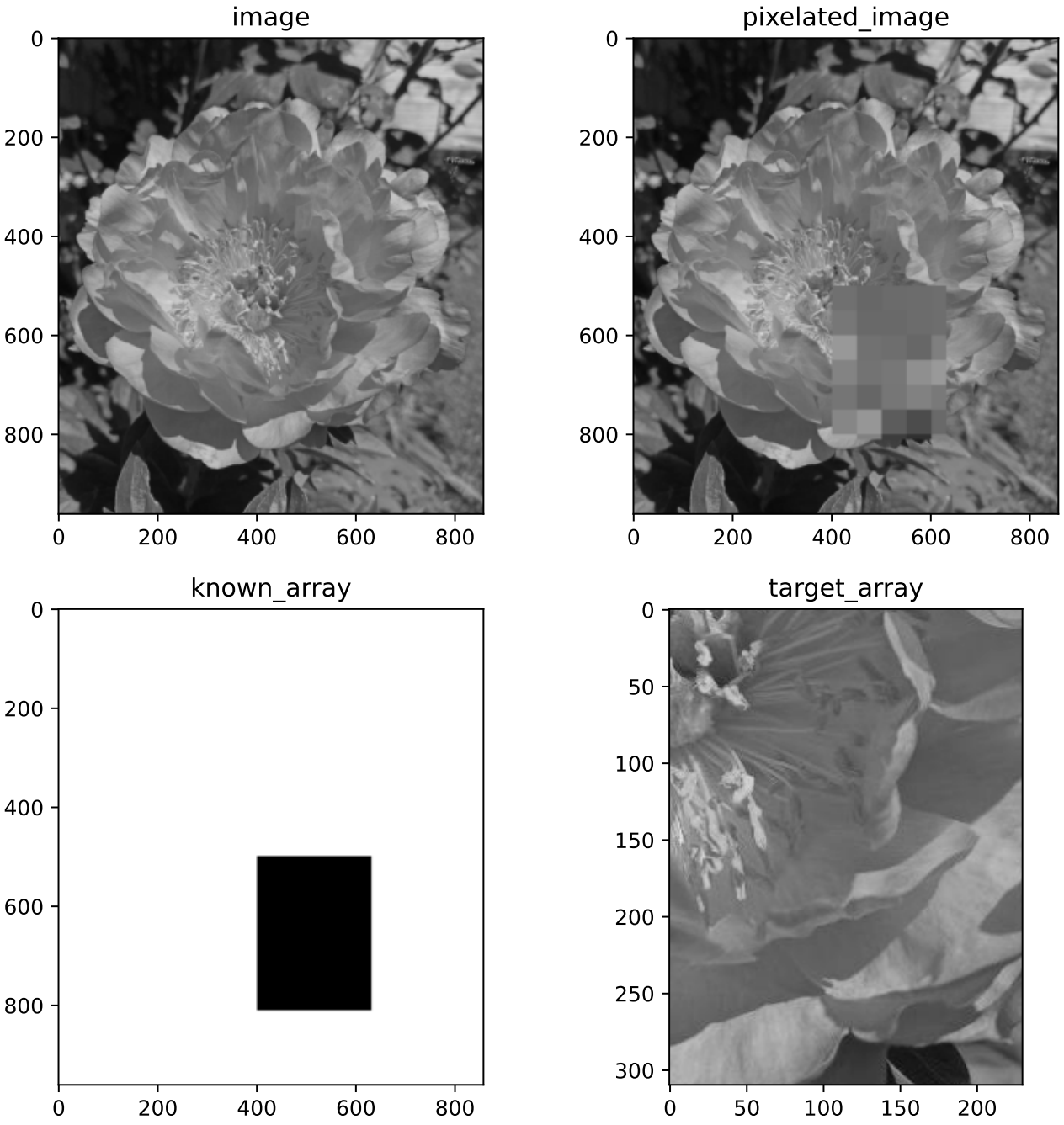

Figure 3: Example return values `pixelated_image`, `known_array` and `target_array` for some given input image with shape `(1, 961, 858)`. The results were obtained with arguments `x=400`, `y=500`, `width=230`, `height=310` and `size=50`. This example also shows shrunk block sizes on the right and bottom of the pixelated area (see `pixelated_image`), since neither `width` nor `height` are evenly divisible by `size`.