Data Loading

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: `a3_ex1.py` 60 Points

Write a class `RandomImagePixelationDataset` that extends `torch.utils.data.Dataset` and is responsible for providing the pixelated images and their additional data which was part of the previous assignment. The class has the following three instance methods:

- ```
  __init__(
      self,
      image_dir,
      width_range: tuple[int, int],
      height_range: tuple[int, int],
      size_range: tuple[int, int],
      dtype: Optional[type] = None
  )
  ```

  `image_dir` specifies the directory where image files with the extension `".jpg"` should be searched for (recursively, i.e., including subdirectories). The found files must be collected using their absolute paths, and the list of these must be sorted afterwards in ascending order.

  `width_range`, `height_range` and `size_range` indicate 2-tuples, where the first entry is the smallest possible value (minimum) and the second entry the largest possible (maximum) value of a range from which a random sample will be chosen (see `__getitem__` below). For each of these ranges, the following checks must be performed: If the minimum value is smaller than 2, a `ValueError` must be raised. If the minimum value is greater than the maximum value, a `ValueError` must be raised.

  `dtype` optionally specifies the data type of the loaded images (see `__getitem__` below).

- `__getitem__(self, index)`

  This method works as follow:

  - Given the specified integer `index`, the `index`-th image from the sorted list of image files (see `__init__` above) must be loaded with `PIL.Image.open`.
  - The image is then stored in a NumPy array using the optionally specified `dtype` (otherwise, the default data type is used).
  - This image array is then transformed into grayscale using the `to_grayscale` method from the previous assignment.
  - Afterwards, again from the previous assignment, the method `prepare_image(image, x, y, width, height, size)` must be called where `x`, `y`, `width`, `height` and `size` are determined as follows: Given a random number generator whose seed must be set to `index`, `width` and `height` are chosen randomly given the respective (inclusive) value ranges `width_range` and `height_range`. If the randomly chosen values should be greater
  ```

than the actual image dimensions, the values are capped/clipped. Then, `x` and `y` are chosen randomly such that they will always fit into the image given the random `width` and `height`, i.e., within the range $[0, \texttt{image\_width} - \texttt{width}]$ and $[0, \texttt{image\_height} - \texttt{height}]$, respectively.

- Lastly, `size` is again chosen randomly given the (inclusive) value range `size_range`.

The method must then return the following 4-tuple: `(pixelated_image, known_array, target_array, image_file)`, where the first three entries are the values returned by `prepare_image`, and the fourth entry is the absolute file path of the image that was loaded.

- `__len__(self)`

  Returns the number of samples, i.e., the number of images that were found in `__init__`.

Example program execution:

```
import matplotlib.pyplot as plt

ds = RandomImagePixelationDataset(
    r"C:\some\path\to\imgs",
    width_range=(50, 300),
    height_range=(50, 300),
    size_range=(10, 50)
)
for pixelated_image, known_array, target_array, image_file in ds:
    fig, axes = plt.subplots(ncols=3)
    axes[0].imshow(pixelated_image[0], cmap="gray", vmin=0, vmax=255)
    axes[0].set_title("pixelated_image")
    axes[1].imshow(known_array[0], cmap="gray", vmin=0, vmax=1)
    axes[1].set_title("known_array")
    axes[2].imshow(target_array[0], cmap="gray", vmin=0, vmax=255)
    axes[2].set_title("target_array")
    fig.suptitle(image_file)
    fig.tight_layout()
    plt.show()
```

Example output (assuming some images in the provided directory):

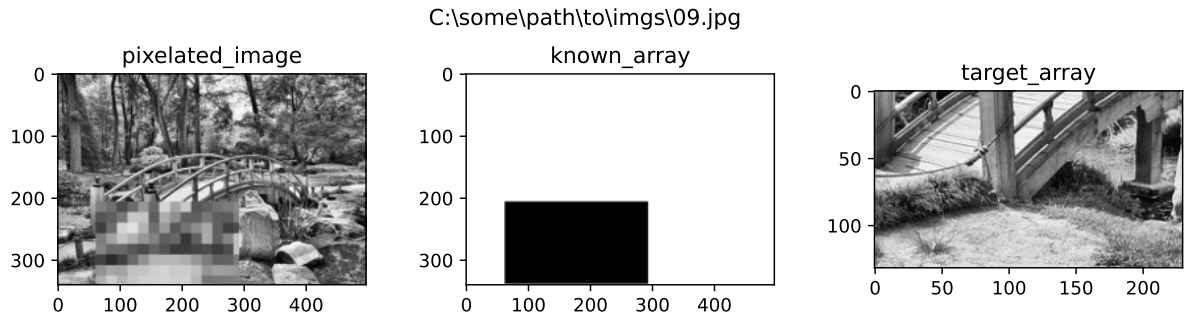

Figure 1: Example visualization of one output produced by the above code.

Exercise 2 – Submission: `a3_ex2.py` 40 Points

Write a function `stack_with_padding(batch_as_list: list)` that can be used as `collate_fn` in a `torch.utils.data.DataLoader`. It must work on samples provided by `RandomImagePixelationDataset` (see exercise above), i.e., 4-tuples of `(pixelated_image, known_array, target_array, image_file)`, as follows:

- Each `pixelated_image` must be stacked according to the maximum shape in the given batch. The maximum shape is determined by the maximum height and maximum width of all batch images, which can be independent from one another, i.e., the maximum values can be from different images. Images that must be extended to this maximum shape (i.e., all smaller images in this batch) must be padded with 0. The stacking dimension must be the first dimension, i.e., the stacked result has the shape `(N, 1, H, W)`, where `N` is the batch size (the number of samples in the given batch), `1` the brightness channel size, and `H` is the maximum height and `W` the maximum width of all batch images. The data type of the stacked result must match the data type of the images. Ultimately, the stacked result must be converted to a PyTorch tensor.
- Each `known_array` must be processed exactly the same as above. The only difference is that images that must be extended to match the maximum shape must be padded with 1 (`True`).
- Each `target_array` must be converted to a PyTorch tensor. The resulting tensors are then stored in a list.
- Each `image_file` must be stored in a list (no conversion is done here).

The function must then return the following 4-tuple: `(stacked_pixelated_images, stacked_known_arrays, target_arrays, image_files)`, where the individual entries are as explained above.

Example program execution:

```
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader
from a3_ex1 import RandomImagePixelationDataset

ds = RandomImagePixelationDataset(
    r"C:\some\path\to\imgs",
    width_range=(50, 300),
    height_range=(50, 300),
    size_range=(10, 50)
)
dl = DataLoader(ds, batch_size=2, shuffle=False, collate_fn=stack_with_padding)
for (stacked_pixelated_images, stacked_known_arrays, target_arrays, image_files) in dl:
    fig, axes = plt.subplots(nrows=dl.batch_size, ncols=3)
    for i in range(dl.batch_size):
        axes[i, 0].imshow(stacked_pixelated_images[i][0], cmap="gray", vmin=0, vmax=255)
        axes[i, 1].imshow(stacked_known_arrays[i][0], cmap="gray", vmin=0, vmax=1)
        axes[i, 2].imshow(target_arrays[i][0], cmap="gray", vmin=0, vmax=255)
    fig.tight_layout()
    plt.show()
```

Example output (assuming some images in the provided directory):

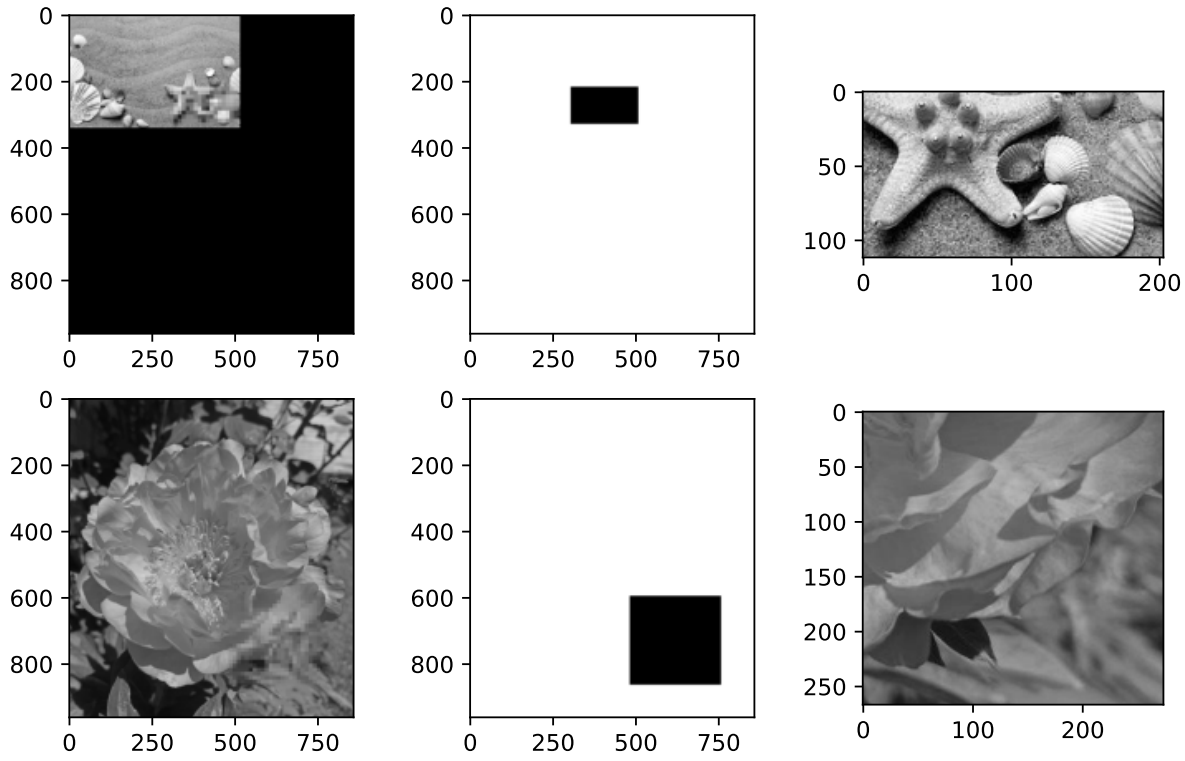

Figure 2: Example visualization of one output produced by the above code using a batch size of 2 (each row represents a single sample of this batch). This shows an extreme case of padding since the first image in the batch with original shape `(1, 340, 513)` is much smaller than the second image with original shape `(1, 961, 858)`. The resulting stacked shape is `(2, 1, 961, 858)` for both the stacked images and stacked known arrays (first and second plot column), the target value shapes are unchanged (third plot column).