Exercise 3: Solution

ImageFolderDataset: __len__()

```
def __len__(self):
    length = None
    ############################################################################
    # TODO:                                                                    #
    # Return the length of the dataset (number of images)                      #
    ############################################################################
    length = len(self.images)
    ############################################################################
    #                            END OF YOUR CODE                              #
    ############################################################################
    return length
```

ImageFolderDataset: __getitem__()

```
def __getitem__(self, index):
    data_dict = None
    ###########################################################################
    # TODO:                                                                   #
    # create a dict of the data at the given index in your dataset            #
    # The dict should be of the following format:                             #
    # {"image": <i-th image>,                                                 #
    # "label": <label of i-th image>}                                         #
    # Hints:                                                                  #
    #   - use load_image_as_numpy() to load an image from a file path         #
    #   - If applicable (Task 4: 'Transforms and Image Preprocessing'),       #
    #     make sure to apply self.transform to the image:                     #
    #     image_transformed = self.transform(image)                           #
    ###########################################################################
    label = self.labels[index]
    path = self.images[index]
    image = self.load_image_as_numpy(path)
    if self.transform is not None:
        image = self.transform(image)
    data_dict = {
        "image": image,
        "label": label,
    }
    ###########################################################################
    #                           END OF YOUR CODE                              #
    ###########################################################################
    return data_dict
```

Hints:

- self.images[index] contains the full name of the image we want to retrieve (we don't want to keep all images in memory at the same time - we only read them when it's required)
- Self.labels[index] contains the label of the image we want to retrieve
- We only apply the transformation if it's not None

RescaleTransform: __call__()

```
def __call__(self, images):
    ###########################################################################
    # TODO:                                                                   #
    # Rescale the given images:                                               #
    #    - from (self._data_min, self._data_max)                              #
    #    - to (self.min, self.max)                                            #
    ###########################################################################
    images = images - self._data_min  # normalize to (0, data_max-data_min)
    images /= (self._data_max - self._data_min)  # normalize to (0, 1)
    images *= (self.max - self.min)  # norm to (0, target_max-target_min)
    images += self.min  # normalize to (target_min, target_max)
    ###########################################################################
    #                            END OF YOUR CODE                             #
    ###########################################################################
    return images
```

compute_image_mean_and_std()

```python
def compute_image_mean_and_std(images):
    """
    Calculate the per-channel image mean and standard deviation of given images
    :param images: numpy array of shape NxHxWxC
        (for N images with C channels of spatial size HxW)
    :returns: per-channels mean and std; numpy array of shape C
    """
    mean, std = None, None
    ################################################################################
    # TODO:                                                                        #
    # Calculate the per-channel mean and standard deviation of the images          #
    # Hint: You can use numpy to calculate mean and standard deviation             #
    ################################################################################
    mean = np.mean(images, axis=(0, 1, 2))
    std = np.std(images, axis=(0, 1, 2))
    ################################################################################
    #                              END OF YOUR CODE                                #
    ################################################################################
    return mean, std
```

Dataloader: __len__()

```
def __len__(self):
    length = None
    ############################################################################
    # TODO:                                                                    #
    # Return the length of the dataloader                                      #
    # Hint: this is the number of batches you can sample from the dataset.     #
    # Don't forget to check for drop last!                                     #
    ############################################################################
    if self.drop_last:
        length = len(self.dataset) // self.batch_size
    else:
        length = int(np.ceil(len(self.dataset) / self.batch_size))
    ############################################################################
    #                             END OF YOUR CODE                             #
    ############################################################################
    return length
```

Dataloader: __iter__()

```
__iter__(self):
########################################################################
# TODO:
# Define an iterable function that samples batches from the dataset.
# Each batch should be a dict containing numpy arrays of length
# batch_size (except for the last batch if drop_last=True)
# Hints:
#   - np.random.permutation(n) can be used to get a list of all
#     numbers from 0 to n-1 in a random order
#   - To load data efficiently, you should try to load only those
#     samples from the dataset that are needed for the current batch.
#     An easy way to do this is to build a generator with the yield
#     keyword, see https://wiki.python.org/moin/Generators
#   - Have a look at the "DataLoader" notebook first. This function is
#     supposed to combine the functions:
#       - combine_batch_dicts
#       - batch_to_numpy
#       - build_batch_iterator
#     in section 1 of the notebook.
########################################################################
def combine_batch_dicts(batch):
    """
    Combines a given batch (list of dicts) to a dict of numpy arrays
    :param batch: batch, list of dicts
        e.g. [{k1: v1, k2: v2, ...}, {k1:, v3, k2: v4, ...}, ...]
    :returns: dict of numpy arrays
        e.g. {k1: [v1, v3, ...], k2: [v2, v4, ...], ...}
    """
    batch_dict = {}
    for data_dict in batch:
        for key, value in data_dict.items():
            if key not in batch_dict:
                batch_dict[key] = []
            batch_dict[key].append(value)
    return batch_dict
```

Hints:

We create two helper functions: one for merging a batch of dictionaries as well as a convenient way to convert those dictionaries to numpy arrays which we will then feed to our networks later.

Dataloader: __iter__()

```
def batch_to_numpy(batch):
    """Transform all values of the given batch dict to numpy arrays"""
    numpy_batch = {}
    for key, value in batch.items():
        numpy_batch[key] = np.array(value)
    return numpy_batch

if self.shuffle:
    index_iterator = iter(np.random.permutation(len(self.dataset)))
else:
    index_iterator = iter(range(len(self.dataset)))

batch = []
for index in index_iterator:
    batch.append(self.dataset[index])
    if len(batch) == self.batch_size:
        yield batch_to_numpy(combine_batch_dicts(batch))
        batch = []

if len(batch) > 0 and not self.drop_last:
    yield batch_to_numpy(combine_batch_dicts(batch))
```

Hints:

- Shuffling is implemented here using numpy's random permutation but there are multiple possible solutions
- We iterate over the dataset and use yield to properly invoke our iterator
- Finally we have to check for the last batch size in order to account for "drop_last".

Questions? Piazza 😊