Numpy Tutorial

Introduction

The purpose of this exercise is to refresh your knowledge of Python and NumPy. We will use numpy functions to create different types of patterns, and implement an image generator class to load and to synthetically augment the data using rigid transformations. If you have never programmed with Python before, please make yourself familiar with the language first, e.g., at https://docs.python.org/3/tutorial/.

For all exercise parts, unit tests are available in the file NumpyTests.py. These tests will help you to assess whether your implementation is correct or not. Note that these unit tests also serve as an extension of slides and description. Many IDEs (e.g., PyCharm) offer the option to run these unittests directly, or you can run via the command line:

```
python NumpyTests.py
```

to run all tests or

```
python NumpyTest.py <TestName>
```

to run one specific test.

1 Array Manipulation Warm-up

1.1 Exercise Skeleton

Each pattern should be implemented as a separate python class and should provide the following methods: a constructor **__init__**(), a method **draw**(), which creates the pattern using numpy functions and a visualization function **show**(). Each pattern has a public member **output**, which is an np.ndarray that stores the respective pattern.
A main script which imports and calls all these classes should also be implemented, which you can use for debugging as well. There are **no loops** needed/allowed for the creation of the patterns in this exercise! Since python is a scripting language, loops would significantly impact the performance. Also get used to proper numpy array indexing and slicing which will be tremendously important for future exercises.

Task:

- Create a file "pattern.py" and implement the classes **Checker** and **Circle** in this file. Note that we do not provide any skeleton here. Also create a file "main.py", which imports all other classes.

- Import numpy for calculation and matplotlib for visualisation using
  `import numpy as np`
  and
  `import matplotlib.pyplot as plt`.
  This is the most common way to import those packages.

Hints:
__init__() is the constructor of the class. Following functions from the cheat sheet might be useful: **np.tile()**, **np.arange()**, **np.zeros()**, **np.ones()**, **np.concatenate()** and **np.expand_dims()**

1.2 Checkerboard

The first pattern to implement is a checkerboard pattern in the class **Checkers** with adaptable tile size and resolution. You might want to start with a fixed tile size and adapt later on. For simplicity we assume that the resolution is divisible by the tile size without remainder.

Task:

- Implement the constructor. It receives two arguments: an integer **resolution** that defines the number of pixels in each dimension, and an integer **tile_size** that defines the number of pixel an individual tile has in each dimension. Store the arguments as public members. Create an additional member variable **output** that can store the pattern.
- Implement the method **draw()** which creates the checkerboard pattern as a numpy array. The tile in the top left corner should be black. In order to avoid truncated checkerboard patterns, make sure your code only allows values for **resolution** that are evenly dividable by 2$\cdot$ **tile_size**. Store the pattern in the public member output and return a copy. Helpful functions for that can be found on the **Deep Learning Cheatsheet** provided.
- Implement the method **show()** which shows the checkerboard pattern with for example **plt.imshow()**. If you want to display a grayscale image you can use **cmap = gray** as a parameter for this function.
- Verify your implementation visually by creating an object of this class in your main script and calling the object's functions.
- Verify your implementation by calling the unit tests with TestCheckers.

Hint: Try to build your checkerboard out of simpler constituents. Think about how tile_size and resolution must relate to each other in order to get a valid checkerboard pattern.

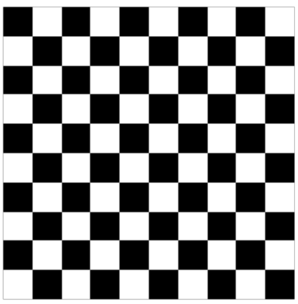

Figure 1: Checkerboard example.

1.3 Circle

The second pattern to implement is a binary circle with a given radius at a specified position in the image. Note that we expect you to use numpy operations to draw this pattern. We do not accept submissions which draw a circle with a single library function call.

Task:

- Implement the constructor. It receives three arguments: An integer **resolution**, an integer **radius** that describes the radius of the circle, and a tuple **position** that describes the position of the circle center in the image.
- Implement the method **draw()** which creates a binary image of a circle as a numpy array. Store the pattern in the public member **output** and return a copy.
- Implement the method **show()** which shows the circle with for example **plt.imshow()**.
- Verify your implementation visually by creating an object of this class in your main script and calling the object's functions.
- Verify your implementation by calling the unit tests with TestCircle.

Hints:
Think of a formula describing the circle with respect to pixel coordinates. Make yourself familiar with np.meshgrid.

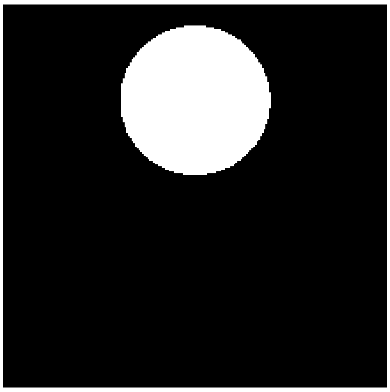

Figure 2: Binary circle example.

1.4 Color Spectrum

The third (optional) pattern to implement is an RGB color spectrum. To enable the corresponding unittest, just go ahead and start implementing. Once a class **Spectrum** is defined in "pattern.py", the corresponding section in the unittests gets activated automatically.

Task:

- Implement the constructor. It receives one parameter: an integer **resolution**.
- Implement the method **draw()** which creates the spectrum in Fig. 3 as a numpy array. Remember that RGB images have 3 channels and that a spectrum consists of rising values across a specific dimension. For each color channel, the intensity minimum and maximum should be 0.0 and 1.0, respectively. Store the pattern in the public member **output** and return a copy. Hint: Particularly take a look into the corners and their color, to figure out the underlying distribution of the channels.
- Implement the method **show()** which shows the RGB spectrum with for example **plt.imshow()**.
- Verify your implementation visually by creating an object of this class in your main script and calling the object's functions.
- Verify your implementation by calling the unit tests with TestSpectrum.

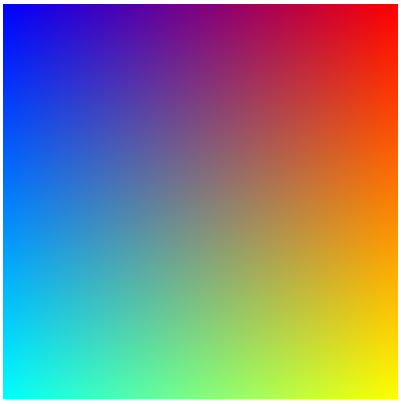

Figure 3: RGB spectrum example.

2 Data Handling Warmup

One of the most important tasks for machine learning is adequate pre-processing and data handling. In the following, we will implement a class that is able to read in a set of images, their associated class labels (stored as a JSON file), and generate batches (subsets of the data) that can be used for training of a neural network.

2.1 Image Generator

Task:

- Implement the class **ImageGenerator** in the file "generator.py".
- Provide a constructor **__init__()** receiving
  1. the path to the directory containing all images **file_path** as a string
  2. the path to the JSON file **label_path** containing the labels again as string
  3. an integer **batch_size** defining the number of images in a batch.
  4. a list of integers defining the desired **image_size** [height,width,channel]
  5. and optional bool flags **rotation**, **mirroring**, **shuffle** which default to False.
- The labels in the JSON file are stored as a dictionary, where the key represents the corresponding filename of the images as a string (e.g. the key '15' corresponds to the image 15.npy) and the value of each key stands for the respective class label encoded as integer. (0 = 'airplane'; 1 = 'automobile'; 2 = 'bird'; 3 = 'cat'; 4 = 'deer'; 5 = 'dog'; 6 = 'frog'; 7 = 'horse'; 8 = 'ship'; 9 = 'truck')
- Provide the method **next()**, which returns one batch of the provided dataset as a tuple (images, labels), where images represents a batch of images and labels an array with the corresponding labels, when called.
- Note: Sometimes the images fed into a neural network are first resized. Therefore, a resizing option should be included within the **next()** method. Do not confuse resizing with reshaping! Resizing usually involves interpolation of data, reshaping is the simple reordering of data. It is allowed to use a library function for resizing (Hint: Have a look into skimage.transforms.resize).
- Note: Make sure all your batches have the same size. If the last batch is smaller than the others, complete that batch by reusing images from the beginning of your training data set.
- Implement the following functionalities for data manipulation and augmentation:

- **shuffle**: If the shuffle flag is True, the order in which the images appear is random. Note: With shuffling, the ImageGenerator must not return duplicates within one epoch (= one run through the whole data set). Only the order is random! → If your index reaches the end of your data during batch creation reset your index to point towards the first elements of your dataset and shuffle your indices again after the epoch.
- **mirroring**: If the mirroring flag is True, randomly mirror the images in the method **next()**.
- **rotation**: If the rotation flag is True, randomly rotate the images by 90, 180 or 270° in the method **next()**
- Note: rotation and mirroring should be applied on an image-to-image basis. Therefore it can happen that your batch contains non-rotated images and rotated ones, side by side.

* Implement a method **class_name(int_label)**, which returns the class name that corresponds to the integer label in the argument **int_label**.
* Implement a method **show()** which generates a batch using **next()** and plots it. Use **class_name()** to obtain the titles for the image plots, as shown in Fig. 4.
* Verify your implementation visually by creating an object of this class in your main script and calling **show()**.
* Verify the correct handling in **next()** by calling the unit tests with TestGen.

Hints:
__init__() is the constructor of the class as previously.
Make sure to handle the data type correctly when you visualize the images (see documentation of plt.imshow).
Check out the methods provided in `np.random` for data augmentation.
Have a look at `plt.add_subplot` to simplify the plot generation.

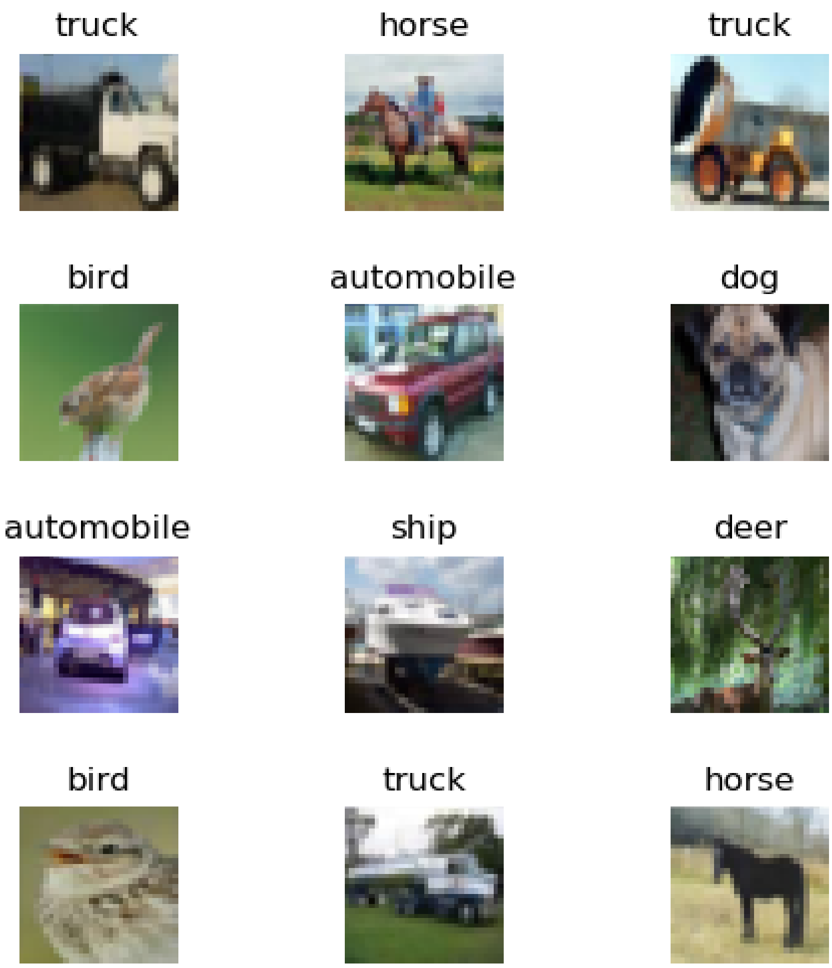

Figure 4: Example image generator output.

3 Test, Debug and Finish

Now we implemented everything.

Task:

Debug your implementation until every test in the suite passes. You can run all tests by providing no commandline parameter. To run the unittests you can either execute them with python in the terminal or with the dedicated unittest environment of PyCharm. We recommend the latter one, as it provides a better overview of all tests. For the automated computation of the bonus points achieved in one exercise, run the unittests with the bonus flag in a terminal, with

```
python3 NumpyTests.py --bonus
```

or set in PyCharm a new "Python" configuration with - -*bonus* as "Parameters". Notice, in some cases you need to set your src folder as "Working Directory". More information about PyCharm configurations can be found here [1].
Make sure you don't forget to upload your submission to StudOn. Use the dispatch tool, which checks all files for completeness and zips the files you need for the upload. Try

```
python3 dispatch.py --help
```

to check out the manual. For dispatching your folder run e.g.

```
python3 dispatch.py -i ./src -o submission.zip
```

and upload the .zip file to StudOn.

[1] https://www.jetbrains.com/help/pycharm/creating-and-editing-run-debug-configurations.html