* 1. Conversion Graph Construction
* Developed a graph-based framework to represent image data types as nodes within a graph structure.
* Encoded various image conversion routines as directional edges in the graph, highlighting the transformation process between different image data types.
* Implemented a single-metadata variation strategy to optimize the graph structure, significantly reducing complexity.

Node Design

Designing nodes within the graph structure involves identifying key metadata that users employ to recognize image data types and facilitate conversion between them. This process was approached through a combination of user interviews, searching package managers like Anaconda and PyPI, and Question-answer utilizing AI-based tools like ChatGPT for asking popular image processing libraries. An analysis of these libraries yielded a summary of the metadata (the columns in the table) describing various data representations, emphasizing the inclusion of semantic information—such as color channel—to ensure clarity. The resulting table serves as the foundation for defining the "image metadata" as a comprehensive descriptor for image data. To reduce duplication information, we pick up “data representation”, “color channel”, “channel order”, “minibatch input”, “data type”, “intensity range”, and “device” as the metadata for describing image data. A combination of these metadata values is termed a "fully expressed image," also the node in the graph.

Table 1 single 2-dimensional image with RGB color space.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| image processing libraries or module | data representation | color channel | channel order | minibatch input | shape (H: height, W, width, C, channel, N: batch size) | (data type, intensity range) | device |
| Scikit-image | NumPy  ndarray | RGB | channel-last | false | (H, W, 3) or  (rows, columns, 3) | (uint8, full range) default  (uint16, full range)  (uint32, full range)  (float, 0 to 1)  (float, -1 to 1)  (int8, full range)  (int16, full range)  (int32, full range) | CPU |
|
|
|
|
|
|
|
| Gray | none | (H, W) or  (rows, columns) |
|
|
|
|
|
|
|
| Opencv-python | NumPy  ndarray | BGR | channel-last | false | (H, W, 3) or  (rows, columns, 3) | (uint8, full range) | CPU |
| Gray | none | (H, W) or  (rows, columns) |
| Pillow, imageio (Pillow backend) | PIL Image | RGB | channel-last | false | (H, W, 3) | (uint8, full range) default  (uint16, full range)  (float32, full range)  (int32, full range) | CPU |
| RGBA | (H, W, 4) |
| Gray | none | (H, W) |
| GrayA(LA) | channel-last | (H, W, 2) |
| scipy.ndimage (2D image) | NumPy  ndarray | RGB | channel-last | false | (H, W, 3) | (uint8, full range) default  (uint16, full range)  (float32, full range)  (int8, full range)  (int16, full range)  (int32, full range) | CPU |
| Gray | none | (H, W) |
| matplotlib(pyplot.imshow) | NumPy  ndarray,  PIL Image | RGB | channel-last | false | (H, W, 3) | (uint8, full range)  (float, 0 to 1) | CPU |
| RGBA | (H, W, 4) |
| Gray | none | (H, W) | (uint8, full range) |
| Pytorch, fastai, torchvision,  Kornia | torch tensor | RGB | channel-first | false,  true | (3, H, W),  (1, 3, H, W) | (uint8, full range)  (float, 0 to 1) (Kornia only support)  (double, full range)  (int8, full range)  (int16, full range)  (int32, full range)  (int64, full range) | CPU, GPU |
| Gray | (1, H, W),  (1, 1, H, W) |
| Tensorflow, keras,  kerasCV | tf tensor | RGB | channel-last | false,  true | (H, W, 3)  (1, H, W, 3) | (uint8, full range)  (uint16, full range)  (uint32, full range)  (uint64, full range)  (float16, full range)  (float32, 0 to 1)  (float64, full range)  (int8, full range)  (int16, full range)  (int32, full range)  (int64, full range) | CPU, GPU |
| Gray | (H, W, 1)  (1, H, W, 1) |

Edge Generation

In our graph-based framework, the directional edges represent transformations between various fully expressed image data types. Each edge encompasses conversion routines that transition from source to target image data, underscoring the one-way nature of these transformations. Given the vast array of potential data conversions, manually creating edges for each format pair is not only time-consuming but also burdensome for developers. This process runs the risk of generating multiple, similar routines for only marginally different conversions, leading to redundant duplication within the system.

To address these challenges, we have implemented a single-metadata variation strategy in creating the edges. This method simply the graph by connecting nodes differing by just one metadata property, substantially reducing the total number of necessary edges. When conversion involves multiple metadata changes, traversing the graph to compile a sequence of single-metadata conversions becomes essential to achieve the desired comprehensive transformation. An optimization algorithm, detailed in the following section, facilitates the identification of the shortest or most efficient conversion paths.

However, the single-metadata variation strategy is not without its limitations. For each edge, users must manually specify both source and target fully expressed image data, along with corresponding conversion routines. This labor-intensive process is prone to errors. Furthermore, as image processing packages continually evolve, ensuring version compatibility of conversion routines presents a significant challenge. Keeping pace with these updates and manually rewriting routines can be overwhelming and may introduce inconsistencies in the system.

To address these issues while maintaining adaptability, we have integrated the factory pattern. This approach automatically generates routines based on user-defined conditions and package versions, thus managing compatibility. It guarantees scalability amidst the emergence of new image data formats and offers flexibility by dynamically generating routines according to specific conditions. This eliminates the need for manually crafting routines for each unique pair of image data. Additionally, we have developed a suite of factors for graph construction and also encourage developers to contribute their conversion methods, further enriching our system capabilities.

One example of the factory function is as follows:

One example is as follows:

|  |
| --- |
| import numpy as np  def example\_factory(source\_metadata, target\_metadata):  def version\_match():  return True  def metadata\_match(source\_metadata, target\_metadata):  if not isinstance(source\_value, np.ndarray):  return False  source\_metadata\_copy = source\_metadata.copy()  target\_metadata\_copy = target\_metadata.copy()  source\_metadata\_copy.pop('color\_channel', None)  target\_metadata\_copy.pop('color\_channel', None)  if source\_metadata\_copy != target\_metadata\_copy:  return False  if (source\_metadata.get('color\_channel') == 'BGR' and  target\_metadata.get('color\_channel') == 'RGB') or \  (source\_metadata.get('color\_channel') == 'RGB' and  target\_metadata.get('color\_channel') == 'BGR'):  return True  return False  if version\_match() and metadata\_match(source\_metadata, target\_metadata):  return f"def convert(value): \n\treturn{source\_value}[:, :, ::-1]"    return None |

this function works for both the color channel conversion (BGR to RGB or RGB to BGR) for the image.

Graph Construction

We design a builder to iterate all possible combinations of fully expressed images and for each combination, test each factory and find the works.

Conversion Routines Generation

A star path-finding algorithm