Creating COCO Dataset

Created	@March 9, 2024 5:40 PM
Tags	

Requirements:

- Anaconda(Jupyter Notebook) : Link - https://www.anaconda.com/download#downloads
- VS Code: Link - https://code.visualstudio.com/download
- GIMP: Link - https://www.gimp.org/downloads/

Process:

Anaconda Part -

- Run the following commands on your anaconda prompt.

```
conda create -n cocosyn python=3.8
conda activate cocosyn
conda install -c conda-forge shapely
```

What is shapely used for?
Shapely is a BSD-licensed Python package for manipulation and analysis of planar geometric objects.

Additional Installations:

```
pip install jupyter
pip install notebook
pip install Pillow
pip install numpy
pip install scikit-image
pip install scipy
pip install tqdm
pip install Shapely
```

- Open jupyter notebook by typying this in anaconda prompt:

```
juypter notebook
```

Using VS Code/ Jupyter Notebook part

I used jupyter notebook for this part.

- Download all the images you need.
- Resize and padd the images by creating the follwing python code file:

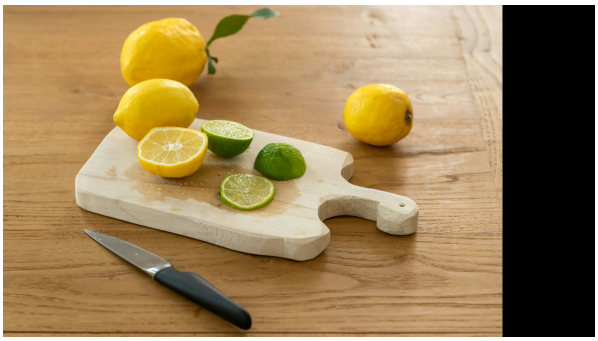
```
import os
import numpy as np
from os import walk
import cv2

def resize_and_pad_images(width, height, input_dir, output_dir)
    print("Working...")

    # Get all the pictures in directory
    images = []
    ext = (".jpeg", ".jpg", ".png")
```

```
    for (dirpath, dirnames, filenames) in walk(input_dir):
        for filename in filenames:
            if filename.endswith(ext):
                images.append(os.path.join(dirpath, filename))

    for image in images:
        img = cv2.imread(image, cv2.IMREAD_UNCHANGED)

        h, w = img.shape[:2]
        pad_bottom, pad_right = 0, 0
        ratio = w / h

        if h > height or w > width:
            # shrinking image algorithm
            print('shrinking')
            interp = cv2.INTER_AREA
        else:
            # stretching image algorithm
            print('stretching')
            interp = cv2.INTER_CUBIC

        w = width
        h = round(w / ratio)
        if h > height:
            h = height
            w = round(h * ratio)
        pad_bottom = abs(height - h)
        pad_right = abs(width - w)

        scaled_img = cv2.resize(img, (w, h), interpolation=inte
        padded_img = cv2.copyMakeBorder(
            scaled_img,0,pad_bottom,0,pad_right,borderType=cv2.B

        if not cv2.imwrite(os.path.join(output_dir, os.path.spli
            raise Exception("Could not write image")
```

```
        else:
            print('Written')


    print("Completed!")

# Example usage:
width = 1280 #use lower values to avoid memory error
height = 720
input_dir = "./images"
output_dir = "./images_resized"

resize_and_pad_images(width, height, input_dir, output_dir)
```

Using GIMP part:

- Open all images in gimp, create n + 1 layers, where n is the number of object to be detected.
- On 1 to n layer, select the object using free select tool and fill color using the bucket fill tool. Fill the n + 1 th layer with black by selecting none object. Then save the file.
- Repeat it for all.
- It should look like this:



Using VS Code

- Create mask_defination.json file:

```
{
  "masks": {
    "images/000001.png": {
      "mask": "masks/000001.png",
      "color_categories": {
        "(0, 255, 0)": { "category": "knife", "super_category":
      }
    },
    "images/000002.png": {
      "mask": "masks/000002.png",
      "color_categories": {
        "(0, 255, 0)": { "category": "knife", "super_category":
      }
    },
    "images/000003.png": {
      "mask": "masks/000003.png",
      "color_categories": {
        "(0, 255, 0)": { "category": "knife", "super_category":
      }
    },
    "images/000004.png": {
      "mask": "masks/000004.png",
      "color_categories": {
        "(0, 255, 0)": { "category": "knife", "super_category":
        "(0, 0, 255)": { "category": "knife", "super_category":
        "(255, 0, 0)": { "category": "knife", "super_category":
      }
    },
    "images/000005.png": {
      "mask": "masks/000005.png",
```

```
      "color_categories": {
        "(0, 255, 0)": { "category": "knife", "super_category":
      }
    }
  },
  "super_categories": {
    "utensil": ["knife"]
  }
}
```

- Create dataset_info.json file:

```
{
  "info": {
    "description": "Test dataset",
    "version": "1",
    "url": "no-url/datasets.com",
    "year": 2024,
    "contributor": "Sayali",
    "date_created": "10/03/2024"
  },
  "license": {
    "url": "no-url/licenses.com",
    "id": 0,
    "name": "testing"
  }
}
```

- Create coco_json_utils.py file:

```
import numpy as np
import json
from pathlib import Path
```

```
from tqdm import tqdm
from skimage import measure, io
from shapely.geometry import Polygon, MultiPolygon
from PIL import Image

class InfoJsonUtils():
    """ Creates an info object to describe a COCO dataset
    """
    def create_coco_info(self, description, url, version, year,
        """ Creates the "info" portion of COCO json
        """
        info = dict()
        info['description'] = description
        info['url'] = url
        info['version'] = version
        info['year'] = year
        info['contributor'] = contributor
        info['date_created'] = date_created

        return info

class LicenseJsonUtils():
    """ Creates a license object to describe a COCO dataset
    """
    def create_coco_license(self, url, license_id, name):
        """ Creates the "licenses" portion of COCO json
        """
        lic = dict()
        lic['url'] = url
        lic['id'] = license_id
        lic['name'] = name

        return lic

class CategoryJsonUtils():
    """ Creates a category object to describe a COCO dataset
```

```
    """
    def create_coco_category(self, supercategory, category_id, n
        category = dict()
        category['supercategory'] = supercategory
        category['id'] = category_id
        category['name'] = name

        return category

class ImageJsonUtils():
    """ Creates an image object to describe a COCO dataset
    """
    def create_coco_image(self, image_path, image_id, image_lice
        """ Creates the "image" portion of COCO json
        """
        # Open the image and get the size
        image_file = Image.open(image_path)
        width, height = image_file.size

        image = dict()
        image['license'] = image_license
        image['file_name'] = image_path.name
        image['width'] = width
        image['height'] = height
        image['id'] = image_id

        return image

class AnnotationJsonUtils():
    """ Creates an annotation object to describe a COCO dataset
    """
    def __init__(self):
        self.annotation_id_index = 0

    def create_coco_annotations(self, image_mask_path, image_id,
        """ Takes a pixel-based RGB image mask and creates COCO
```

```
        Args:
            image_mask_path: a pathlib.Path to the image mask
            image_id: the integer image id
            category_ids: a dictionary of integer category ids l
                e.g. {'(255, 0, 0)': {'category': 'owl', 'super_
        Returns:
            annotations: a list of COCO annotation dictionaries
            be converted to json. e.g.:
            {
                "segmentation": [[101.79,307.32,69.75,281.11,..
                "area": 51241.3617,
                "iscrowd": 0,
                "image_id": 284725,
                "bbox": [68.01,134.89,433.41,174.77],
                "category_id": 6,
                "id": 165690
            }
        """
        # Set class variables
        self.image_id = image_id
        self.category_ids = category_ids

        # Make sure keys in category_ids are strings
        for key in self.category_ids.keys():
            if type(key) is not str:
                raise TypeError('category_ids keys must be strin
            break

        # Open and process image
        self.mask_image = Image.open(image_mask_path)
        self.mask_image = self.mask_image.convert('RGB')
        self.width, self.height = self.mask_image.size

        # Split up the multi-colored masks into multiple 0/1 bi
        self._isolate_masks()
```

```
        # Create annotations from the masks
        self._create_annotations()

        return self.annotations

    def _isolate_masks(self):
        # Breaks mask up into isolated masks based on color

        self.isolated_masks = dict()
        for x in range(self.width):
            for y in range(self.height):
                pixel_rgb = self.mask_image.getpixel((x,y))
                pixel_rgb_str = str(pixel_rgb)

                # If the pixel is any color other than black, ad
                if not pixel_rgb == (0, 0, 0):
                    if self.isolated_masks.get(pixel_rgb_str) is
                        # Isolated mask doesn't have its own ima
                        # with 1-bit pixels, default black. Make
                        # padding on each edge to allow the cont
                        # when shapes bleed up to the edge
                        self.isolated_masks[pixel_rgb_str] = Ima

                    # Add the pixel to the mask image, shifting
                    self.isolated_masks[pixel_rgb_str].putpixel(

    def _create_annotations(self):
        # Creates annotations for each isolated mask

        # Each image may have multiple annotations, so create an
        self.annotations = []
        for key, mask in self.isolated_masks.items():
            annotation = dict()
            annotation['segmentation'] = []
            annotation['iscrowd'] = 0
            annotation['image_id'] = self.image_id
```

```
            if not self.category_ids.get(key):
                print(f'category color not found: {key}; check t
                continue
            annotation['category_id'] = self.category_ids[key]
            annotation['id'] = self._next_annotation_id()

            # Find contours in the isolated mask
            contours = measure.find_contours(mask, 0.5, positive

            polygons = []
            for contour in contours:
                # Flip from (row, col) representation to (x, y)
                # and subtract the padding pixel
                for i in range(len(contour)):
                    row, col = contour[i]
                    contour[i] = (col - 1, row - 1)

                # Make a polygon and simplify it
                poly = Polygon(contour)
                if (poly.area > 16): # Ignore tiny polygons
                    poly = poly.simplify(1.0, preserve_topology=
                    polygons.append(poly)
                    segmentation = np.array(poly.exterior.coords
                    annotation['segmentation'].append(segmentati

            if len(polygons) == 0:
                # This item doesn't have any visible polygons, i
                # (This can happen if a randomly placed foregrou
                #  by other foregrounds)
                continue

            # Combine the polygons to calculate the bounding box
            multi_poly = MultiPolygon(polygons)
            x, y, max_x, max_y = multi_poly.bounds
            self.width = max_x - x
            self.height = max_y - y
```

```
            annotation['bbox'] = (x, y, self.width, self.height
            annotation['area'] = multi_poly.area

            # Finally, add this annotation to the list
            self.annotations.append(annotation)

    def _next_annotation_id(self):
        # Gets the next annotation id
        # Note: This is not a unique id. It simply starts at 0 a

        a_id = self.annotation_id_index
        self.annotation_id_index += 1
        return a_id

class CocoJsonCreator():
    def validate_and_process_args(self, args):
        """ Validates the arguments coming in from the command l
            initial processing
        Args:
            args: ArgumentParser arguments
        """
        # Validate the mask definition file exists
        mask_definition_file = Path(args.mask_definition)
        if not (mask_definition_file.exists and mask_definition_
            raise FileNotFoundError(f'mask definition file was n

        # Load the mask definition json
        with open(mask_definition_file) as json_file:
            self.mask_definitions = json.load(json_file)

        self.dataset_dir = mask_definition_file.parent

        # Validate the dataset info file exists
        dataset_info_file = Path(args.dataset_info)
        if not (dataset_info_file.exists() and dataset_info_file
            raise FileNotFoundError(f'dataset info file was not
```

```
        # Load the dataset info json
        with open(dataset_info_file) as json_file:
            self.dataset_info = json.load(json_file)

        assert 'info' in self.dataset_info, 'dataset_info JSON w
        assert 'license' in self.dataset_info, 'dataset_info JSO

    def create_info(self):
        """ Creates the "info" piece of the COCO json
        """
        info_json = self.dataset_info['info']
        iju = InfoJsonUtils()
        return iju.create_coco_info(
            description = info_json['description'],
            version = info_json['version'],
            url = info_json['url'],
            year = info_json['year'],
            contributor = info_json['contributor'],
            date_created = info_json['date_created']
        )

    def create_licenses(self):
        """ Creates the "license" portion of the COCO json
        """
        license_json = self.dataset_info['license']
        lju = LicenseJsonUtils()
        lic = lju.create_coco_license(
            url = license_json['url'],
            license_id = license_json['id'],
            name = license_json['name']
        )
        return [lic]

    def create_categories(self):
        """ Creates the "categories" portion of the COCO json
```

```
        Returns:
            categories: category objects that become part of the
            category_ids_by_name: a lookup dictionary for catego
                on the name of the category
        """
        cju = CategoryJsonUtils()
        categories = []
        category_ids_by_name = dict()
        category_id = 1 # 0 is reserved for the background

        super_categories = self.mask_definitions['super_categori
        for super_category, _categories in super_categories.iter
            for category_name in _categories:
                categories.append(cju.create_coco_category(super
                category_ids_by_name[category_name] = category_i
                category_id += 1

        return categories, category_ids_by_name

    def create_images_and_annotations(self, category_ids_by_name
        """ Creates the list of images (in json) and the annotat
            image for the "image" and "annotations" portions of
        """
        iju = ImageJsonUtils()
        aju = AnnotationJsonUtils()

        image_objs = []
        annotation_objs = []
        image_license = self.dataset_info['license']['id']
        image_id = 0

        mask_count = len(self.mask_definitions['masks'])
        print(f'Processing {mask_count} mask definitions...')

        # For each mask definition, create image and annotations
        for file_name, mask_def in tqdm(self.mask_definitions['m
```

```
            # Create a coco image json item
            image_path = Path(self.dataset_dir) / file_name
            image_obj = iju.create_coco_image(
                image_path,
                image_id,
                image_license)
            image_objs.append(image_obj)

            mask_path = Path(self.dataset_dir) / mask_def['mask

            # Create a dict of category ids keyed by rgb_color
            category_ids_by_rgb = dict()
            for rgb_color, category in mask_def['color_categorie
                category_ids_by_rgb[rgb_color] = category_ids_by
            annotation_obj = aju.create_coco_annotations(mask_pa
            annotation_objs += annotation_obj # Add the new anno
            image_id += 1

        return image_objs, annotation_objs

    def main(self, args):
        self.validate_and_process_args(args)

        info = self.create_info()
        licenses = self.create_licenses()
        categories, category_ids_by_name = self.create_categorie
        images, annotations = self.create_images_and_annotations

        master_obj = {
            'info': info,
            'licenses': licenses,
            'images': images,
            'annotations': annotations,
            'categories': categories
        }
```

```
        # Write the json to a file
        output_path = Path(self.dataset_dir) / 'knife_dataset_co
        with open(output_path, 'w+') as output_file:
            json.dump(master_obj, output_file)

        print(f'Annotations successfully written to file:\n{outp

if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description="Generate COCO

    parser.add_argument("-md", "--mask_definition", dest="mask_d
        help="path to a mask definition JSON file, generated by
    parser.add_argument("-di", "--dataset_info", dest="dataset_i
        help="path to a dataset info JSON file")

    args = parser.parse_args()

    cjc = CocoJsonCreator()
    cjc.main(args)
```

- Run the following ommand on anaconda prompt:

```
python coco_json_utils.py -md mask_defination.json -di dataset_i
```

As a result a knife_dataset_coco.json file gets created.