Object detection

Object detection models identify something in an image, and object detection datasets are used for applications such as autonomous driving and detecting natural hazards like wildfire. This guide will show you how to apply transformations to an object detection dataset following the tutorial from Albumentations.

To run these examples, make sure you have up-to-date versions of albumentations and cv2 installed:

```
pip install -U albumentations opencv-python
```

In this example, you'll use the `cppe-5` dataset for identifying medical personal protective equipment (PPE) in the context of the COVID-19 pandemic.

Load the dataset and take a look at an example:

```
>>> from datasets import load_dataset

>>> ds = load_dataset("cppe-5")
>>> example = ds['train'][0]
>>> example
{'height': 663,
 'image': <PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=943x663 at 0x7FC3DC756250>,
 'image_id': 15,
 'objects': {'area': [3796, 1596, 152768, 81002],
  'bbox': [[302.0, 109.0, 73.0, 52.0],
   [810.0, 100.0, 57.0, 28.0],
   [160.0, 31.0, 248.0, 616.0],
   [741.0, 68.0, 202.0, 401.0]],
  'category': [4, 4, 0, 0],
  'id': [114, 115, 116, 117]},
 'width': 943}
```

The dataset has the following fields:

- `image`: PIL.Image.Image object containing the image.

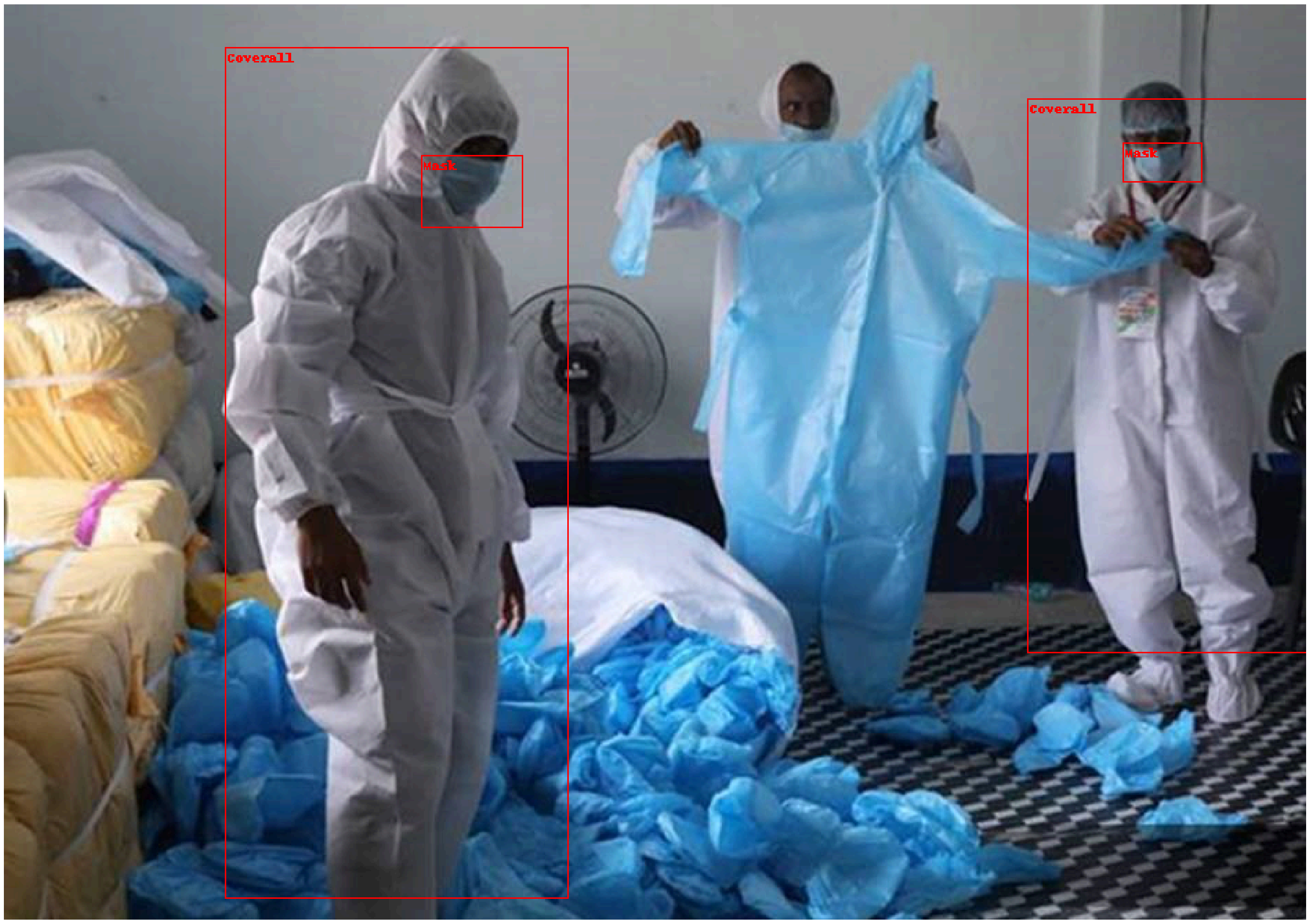
- `image_id`: The image ID.
- `height`: The image height.
- `width`: The image width.
- `objects`: A dictionary containing bounding box metadata for the objects in the image:
  - `id`: The annotation id.
  - `area`: The area of the bounding box.
  - `bbox`: The object's bounding box (in the coco format).
  - `category`: The object's category, with possible values including `Coverall (0)`, `Face_Shield (1)`, `Gloves (2)`, `Goggles (3)` and `Mask (4)`.

You can visualize the `bboxes` on the image using some internal torch utilities. To do that, you will need to reference the ClassLabel feature associated with the category IDs so you can look up the string labels:

```
>>> import torch
>>> from torchvision.ops import box_convert
>>> from torchvision.utils import draw_bounding_boxes
>>> from torchvision.transforms.functional import pil_to_tensor, to_pil_image

>>> categories = ds['train'].features['objects'].feature['category']

>>> boxes_xywh = torch.tensor(example['objects']['bbox'])
>>> boxes_xyxy = box_convert(boxes_xywh, 'xywh', 'xyxy')
>>> labels = [categories.int2str(x) for x in example['objects']['category']]
>>> to_pil_image(
...     draw_bounding_boxes(
...         pil_to_tensor(example['image']),
...         boxes_xyxy,
...         colors="red",
...         labels=labels,
...     )
... )
```



With `albumentations`, you can apply transforms that will affect the image while also updating the `bboxes` accordingly. In this case, the image is resized to (480, 480), flipped horizontally, and brightened.

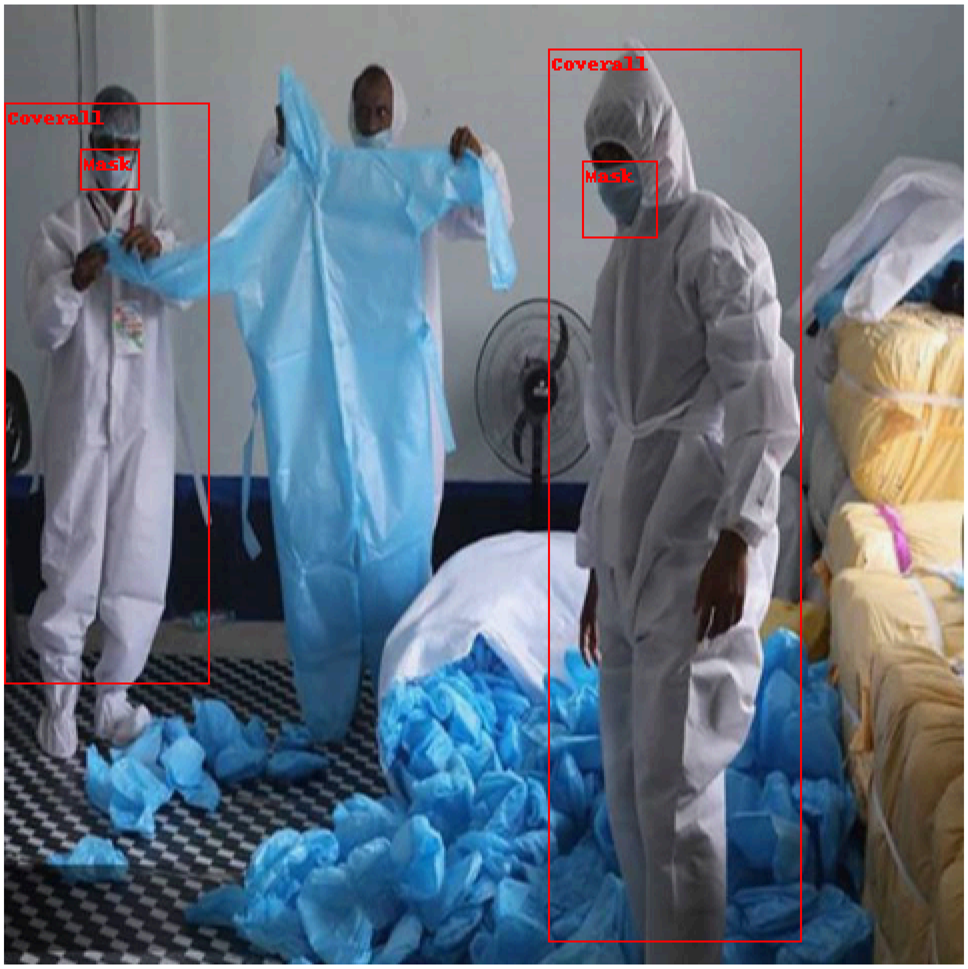
```
>>> import albumentations
>>> import numpy as np

>>> transform = albumentations.Compose([
...     albumentations.Resize(480, 480),
...     albumentations.HorizontalFlip(p=1.0),
...     albumentations.RandomBrightnessContrast(p=1.0),
... ], bbox_params=albumentations.BboxParams(format='coco',  label_fields=['category']))

>>> image = np.array(example['image'])
>>> out = transform(
...     image=image,
...     bboxes=example['objects']['bbox'],
...     category=example['objects']['category'],
... )
```

Now when you visualize the result, the image should be flipped, but the `bboxes` should still be in the right places.

```
>>> image = torch.tensor(out['image']).permute(2, 0, 1)
>>> boxes_xywh = torch.stack([torch.tensor(x) for x in out['bboxes']])
>>> boxes_xyxy = box_convert(boxes_xywh, 'xywh', 'xyxy')
>>> labels = [categories.int2str(x) for x in out['category']]
>>> to_pil_image(
...     draw_bounding_boxes(
...         image,
...         boxes_xyxy,
...         colors='red',
...         labels=labels
...     )
... )
```



Create a function to apply the transform to a batch of examples:

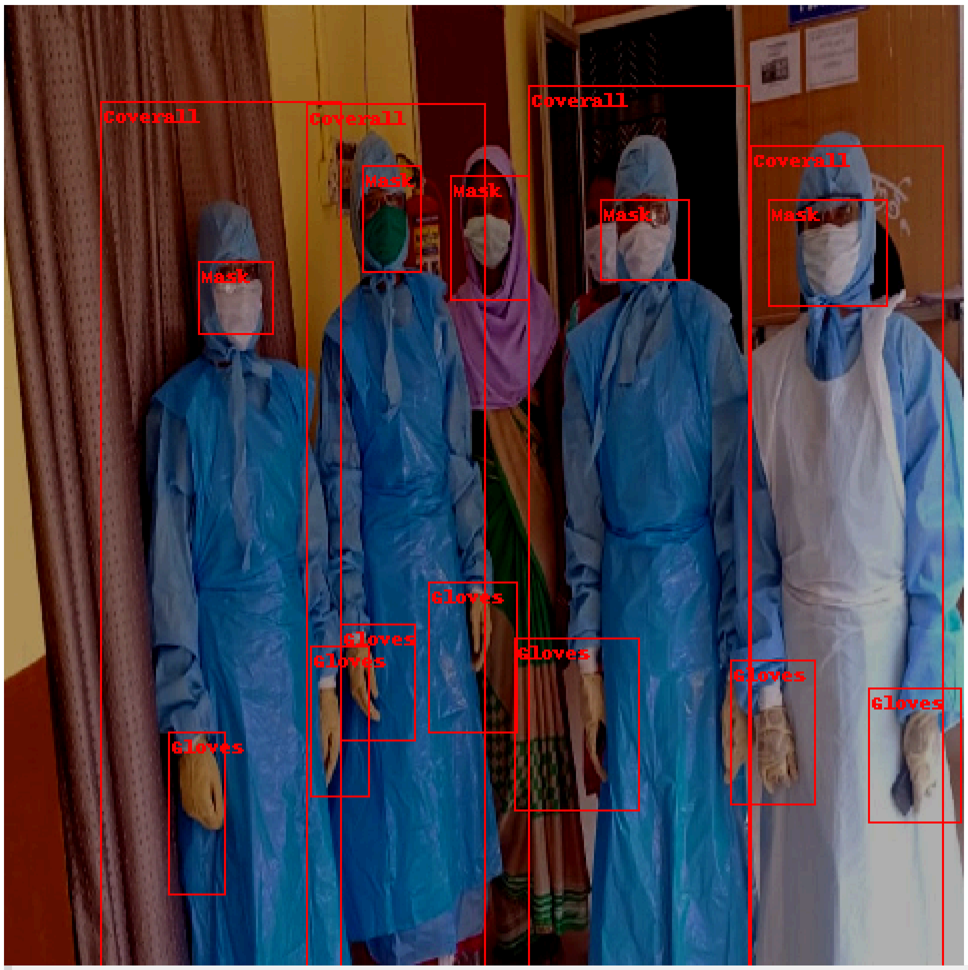
```
>>> def transforms(examples):
...     images, bboxes, categories = [], [], []
...     for image, objects in zip(examples['image'], examples['objects']):
...         image = np.array(image.convert("RGB"))
...         out = transform(
...             image=image,
...             bboxes=objects['bbox'],
...             category=objects['category']
...         )
...         images.append(torch.tensor(out['image']).permute(2, 0, 1))
...         bboxes.append(torch.tensor(out['bboxes']))
...         categories.append(out['category'])
...     return {'image': images, 'bbox': bboxes, 'category': categories}
```

Use the set_transform() function to apply the transform on-the-fly which consumes less disk space. The randomness of data augmentation may return a different image if you access the same example twice. It is especially useful when training a model for several epochs.

```
>>> ds['train'].set_transform(transforms)
```

You can verify the transform works by visualizing the 10th example:

```
>>> example = ds['train'][10]
>>> to_pil_image(
...     draw_bounding_boxes(
...         example['image'],
...         box_convert(example['bbox'], 'xywh', 'xyxy'),
...         colors='red',
...         labels=[categories.int2str(x) for x in example['category']]
...     )
... )
```



> [!TIP]

> Now that you know how to process a dataset for object detection, learn
> how to train an object detection model
> and use it for inference.