detectron2 tutorial translation

杨资璋（翻译）

2021 年 11 月 5 日

1 数据集

这篇文档解释了数据集 API(DatasetCatalog, MetadataCatalog) 是如何工作的，以及如何使用它们来添加自定义的数据库

在 detectron2 中集成支持的数据里列在了这里。如果你想使用自定义的数据集同时复用 detectron2 的数据加载器，你需要做的是：

- 注册你的数据集 (也就是，告诉 detectron2 如何得到你的数据集)
- 以及可选的，为你的数据集注册元数据

接下来，我们详细解释以上两个概念

1.1 注册数据集

为了让 detectron2 知道如何得到名为 “my_dataset” 的数据集，用户需要实现一个返回数据集中的物体的函数并告诉 detectron2 这个函数：

```
def my_dataset_function():
...
return list[dict] in the following format
from detectron2.data import DatasetCatalog
DatasetCatalog.register("my_dataset", my_dataset_function)
# later, to access the data:
data: List[Dict] = DatasetCatalog.get("my_dataset")
```

这里，这个代码段将名为 “my_dataset” 的数据集和返回数据的函数关联了起来。如果这个函数被多次使用，那么它必须按照相同的顺序返回相同的数据。知道进程结束，这个注册一直有效。

这个函数可以做任意的事情并应该返回list[dict]形式的数据，每一个 dict 应该是如下的一种格式：

- 下面将要描述的 detectron2 的标准数据集字典。这将会使它可以和 detectron2 的集成支持特性一同工作，所以当可行时，这是推荐的方式
- 任何自定义的格式。你也可以返回任意自定格式的字典，例如为新任务添加额外的键。这是你将需要在下游任务中合适地处理它们。下面将会展示更多细节。

1.1.1 标准数据集字典

对于标准任务（实例检测、实例/语义/全景分割和关键点检测），我们以类似于 COCO 标注的规范将原本的数据集载入list[dict]。这是我们对于数据集的标准表示。

每一个字典包含一张图片的信息。字典可能有如下的域，以及要求的域随着数据加载器和任务的需要而变换（更多见下表）。

- file_name：图片文件的完全路径
- height, width：整数。图片的形状
- image_id：字符串或者整数。可以识别图片的唯一 id。许多评价者为了识别图片可能会要求，但是数据集也可以处于不同的目的使用它
- annotations(list[dict])：实例检测/分割或者关键点检测任务要求这个域。每一个字典对应于图片中一个实例的标注，可能包含如下键：
  - bbox(list[float])：表示实例边界框的包含四个数字的列表
  - bbox_mode(int)：边界框的格式。它一定是structrues.BoxMode中的一员。目前支持：BoxMode.XYXY_ABS和BoxMode.XYWH_ABS
  - category_id(int)：一个 [0, num_categories-1] 中的整数，表示类别标签。如果需要，num_categories 被保留用来表示 “背景” 类别。
  - segmentation(list[list[float]] 或者 dict)：实例的分割掩码
    - 如果是list[list[float]]形式，它表示一个多边形列表，其中每一个多边形表示物体的连接的组分。每一个list[float]是一个

格式为[x1, y1, ..., xn, yn]（n 3）的简单多边形。其中 Xs 和 Ys 是单位像素下的绝对坐标。

* 如果是dict形式，它表示

— keypoints

- sem_seg_file_name
- pan_seg_file_name
- segments_info

— id

— category_id

— iscrowd

1.1.2 为了新任务自定义数据集字典

在你的数据集函数返回的list[dict]中，这个字典也可以包含**任意自定义的数据**。这对于需要使用没有被标准数据字典包含的额外信息的新任务有帮助。在这种情况下，你需要保证下游代码可以正确处理你的数据。这经常要求为数据加载器写一个新的mapper（见使用自定义数据加载器）。

当设计一个自定义格式时，注意所有的字典都储存在内存中（有时被序列化并有多份拷贝）。为了节约内存，每个字典应该包含关于每个样本少但是充分的信息，例如文件名和标注。加载全部样本通常在数据加载器中发生。

对于在整个数据集中共享的属性，使用Metadata（见后文）。为了避免额外的内存，不要将这种信息存在每一个样本中。

1.2 数据集的元数据

每个数据集斗鱼某些元数据向关联,可以通过MetadataCatalog.get(dataset_name).some_metadata访问。元数据是包含整个数据集共享的信息的键值对映射，通常被用来解释数据集中有什么，例如，类别的名字、类别的颜色和根文件等等。这些信息对于数据增强、评估、可视化和日志等很有用。元数据的结构取决于对应下游代码中的需要。

如果你通过DatasetCatalog.register注册了一个新数据集，为了附能一些需要元数据的特性，你可能也想通过MetadataCatalog.get(dataset_name).some_key = some_value来添加它对应的元数据。你可以这样做(使用元数据键“thing_classes”作为例子)：

```
from detectron2.data import MetadataCatalog
MetadataCatalog.get("my_dataset").thing_classes = ["person", "dog"]
```

这里是使用 detectron2 内建特性的元数据列表。如果你在没有这些元数据的情况下添加了你自己的数据集，某些特性可能不可用：

- thing_classes
- thing_colors
- stuff_classes
- ignore_label
- keypoint_names
- keypoint_flip_map
- keypoint_connection_rules

一些额外的为了评估某些数据集（例如 COCO）的特定元数据

- thing_dataset_id_to_contiguous_id
- stuff_dataset_id_to_contiguous_id
- json_file
- panoptic_root和panoptic_json
- evaluator_type

1.3 注册一个 COCO 格式的数据集

如果你的实例层级（检测、分割和关键点）数据集已经是 COCO 格式下的 json 文件，这个数据集以及它关联的元数据可以如下轻易注册：

```
from detectron2.data.datasets import register_coco_instances
register_coco_instances("my_dataset", {}, "json_annotation.json", "path/to/
    image/dir")
```

如果你的数据集是 COCO 的格式但是需要进一步处理，或者有额外的自定义标注，load_coco_json函数可能会有用。

1.4 为了新的数据集更新配置

一旦你注册了数据集，你可以在cfg.DATASETS.TRAIN/TEST中使用数据集的名字。这里还有一些其他的你为了在新的数据集上训练或者评测可能想要改变的配置:

- MODEL.ROI_HEAD.NUM_CLASSES和MODEL.RETINANET.NUM_CLASSES
- MODEL.ROI_KEYPOINT_HEAD.NUM_KEYPOINTS
- MODEL>SEM_SEG_HEAD.NUM_CLASSES
- TEST.DETECTIOND_PER_IMAGE
- DATASETS.PROPOSAL_FILES_TRAIN\TEST

新的模型经常有它们自己类似的需要更改配置。

2 数据加载器

数据加载器是为模型提供数据的模块。一个数据加载器通常（但并不是一直）使用来自数据集的未加工信息，并且将它们处理为模型需要的格式

2.1 现有的数据加载器是如何工作的

detectron2 包含一个内建的数据加载流水线。理解它是如何工作是好的，万一你需要自己写一个呢。

detectron2 提供了两个函数build_detection_train/test_loader来从给定的配置创建数据加载器。这里介绍build_detection_train/test_loader是如何工作的:

- 它接收一个已经注册数据集的名字（例如，"coco_2017_train"）并加载一个以轻量化格式表示数据集项的list[dict]。这些数据集项并没有准备好被模型使用（例如，图片还没有被载入内存、随机的数据增强还没有被应用等等）。数据集格式和数据集注册的细节可以在 [这

里](https://detectron2.readthedocs.io/en/latest/tutorials/datasets.html)被找到。

- 列表中的每一个 dict 都会被函数（“mapper”）映射：
  - 用户可以通过指定`build_detection_train/test_loader`中的 “mapper” 参数来自定义这个映射函数。默认的 mapper 是DatasetMapper
  - mapper 的输出格式可以使任意的，只要它被这个数据加载器的消费者（通常是模型）接受。默认 mapper 的输出，在 batching 后，遵循Use Model中指出的默认的模型输入格式。
  - mapper 的作用是将数据集项的轻量化表示转化为准备好模型使用的格式（包括，例如读入图片、进行随机的数据增强和转化为 torch 张量）。如果你想要对数据进行自定义的转化，你经常想要写一个自定义的 mapper
- mapper 的输出是 batched（简单的放入一个列表中）
- 这个 batched 数据就是数据加载器的输出。一般来说，也是`model.forward()`的输入

2.2 写一个自定义的数据加载器

使用不同 mapper 的`build_detection_train/test_loader`适用大部分自定义数据加载的情况。例如，如果你为了训练想将所有的图片裁剪为固定的尺寸，使用：

```
import detectron2.data.transforms as T
from detectron2.data import DatasetMapper
dataloader = build_detection_train_loader(cfg, mapper=DatasetMapper(cfg,
  is_train=True, augmentations=[T.Reszie((800, 800))]))
```

如果默认DatasetMapper的参数没有提供你需要的，你可以写一个自己的 mapper 函数并使用它，例如：

```
from detectron2.data import detection_utils as utils
def mapper(dataset_dict):
    dataset_dict = copy.deepcopy(dataset_dict)
    image = utils.read_image(dataset_dict["file_name"], format="BGR")
    auginput = T.AugInput(image)
    transfrom = T.Resize((800, 800))(auginput)
    image = torch.from_numpy(auginput.image.transpose(2, 0, 1))
```

```
        annos = [
            utils.transform_instance_annotations(annotation, [transform],
    image.shape[1:])
            for annotation in dataset_dict.pop("annotations")
        ]
        return {
            "image":image,
            "instances": utils.annotations_to_instances(annos, image.shape
    [1:])
        }
    dataloader = build_detection_train_loader(cfg, mapper=mapper)
```

如果你不仅想改变 mapper（例如，为了实现不同的采样或者 batching 逻辑），那么build_detection_train/test_loader不再适用，你需要写一个不同的数据加载器。数据加载器就是一个生产模型接受格式的 Python 的迭代器。你可以通过任何你喜欢的工具实现它。

无论实现什么,我们推荐你去查看API documentation of detectron2.data来学习更多关于这些函数的 API。

2.3 使用一个自定义的数据加载器

如果你使用DefaultTrainer，你可以重写它的build_train/test_loader来使用你自己的数据加载器。例子可见deeplab dataloader

如果你要写自己的训练循环，你可以轻易的插入你的数据加载器

3 数据增强

增强是训练的重要部分。detectron2 的数据增强系统意在达到如下目标:

- 允许同时增强多个数据类型（例如，和图片一起的边界框和掩码）
- 允许应用一系列静态声明的增强
- 允许为增强添加自定义的新数据类型（旋转边界框，视频片段等等）
- 处理和操作由增强应用的操作

前两个特性包含了大部分通产使用的情况，并且在例如albumentations的其他库中也可用。支持其他特性为 detectron2 的增强 API 添加了一些优势，这也是我们将会在这个教程中解释的

这个教程聚焦于当写新的数据加载器时如何使用增强，以及如何写新的增强。如果你使用 detctron2 中的默认数据加载器，正如Dataloader tutorial中解释的，它已经支持接收一个用户提供的自定义增强列表

3.1 基本用法

特征 1 和特征 2 的基本用法如下：

```
from detectron2.data import transforms as T
# 定义一个增强序列
augs = T.AugmentationList([
    T.RandomBrightness(0.9, 1.1),
    T.RandomFlip(prob=0.5),
    T.RandomCrop("absolute", (640, 640))
]) # 类型：T.Augmentation
# 定义增强输入
input = T.AugInput(image, boxes=boxes, sem_seg=sem_seg)
# 应用增强
transform = augs(input)
image_transfomed = input.image # 新图片
sem_seg_transformed = input.sem_seg # 新语义分割

# 对于其他需要被一起增强的额外数据，使用transform，例如：
image2_transformed = transform.apply_image(mage2)
polygons_transformed = transform.apply_polygons(polygons)
```

这里涉及三个基本概念：

- T.Augmentation 定义了需改输入的**政策**
  - 它的`__call__(AugInput) -> Transform`通过 in-place 的方式增强数据，并且返回应用的操作
- T.Transform 实现了真正的变形数据的操作
  - 它有例如`apply_image`和`apply_coords`的方法来定义如何为每一种数据类型变形
- T.AugInput 存储了`T.Augmentation`需要的输入以及它们应该如何变形。一些高阶用法需要这个概念。如上所示，由于没在`T.AugInput`中的额外数据可以通过使用返回的`transform`增强，所以直接使用这个类对于所有常用情形应该是足够的。

3.2 写新的增强

大部分二维增强只需要知道数据图像。这样的增强可以如下轻易实现:

```
class MyColorAugmentation(T.Augmentation):
    def get_transform(self, image):
        r = np.random.rand(2)
        return T.ColorTransform(lambda x: x * r[0] + r[1] * 10)

class MyCustonResize(T.Augmentation):
    def get_transform(self.image):
        old_h, old_w = image.shape[:2]
        new_h, new_w = int(old_h * np.random.rand()), int(old_w * 1.5)
        return T.ResizeTransform(old_h, old_w, new_h, new_w)

augs = MyCustomResize()
transform = augs(input)
```

除了图片，只要他们是函数签名的一部分，给定`AugInput`的任意属性都可以被使用，例如:

```
class MyCustomCrop(T.Augmentation):
    def get_transform(self, image, sem_seg):
        # decide where to crop using both image and sem_seg
        return T.CropTransform(...)

augs = MyCustomCrop()
assert hasattr(input, "image") and hasattr(input, "sem_seg")
transform = augs(input)
```

新的变形操作也可以通过继承T.Transform来添加

3.3 高级用法

我们提供了一些我们的系统使能的高级用法的例子。虽然对于标准使用的情形来说修改它们是不必要的，但是这些选项可能对于新研究来说很有趣。

3.3.1 自定义变形策略

detectron2 的`Augmentation`并不是仅仅返回增强的数据,而是返回`T.Transform`形式的**操作**。这允许用户对数据使用自定义的变形策略。我们使用关键点数据作为例子。

关键点是 (x, y) 坐标，但是由于它们携带的语义信息，所以增强它们并不容易。只有用户知道这样的含义，因此用户想根据返回的transform来手动增强它们。例如，当一个图片是水平翻转的，我们希望将关键点标注“左眼”和“右眼”对换。这可以通过如下代码实现（这被 detectron2 的默认数据加载器包含在内）：

```
# augs和input在之前的例子中定义
transfom = augs(input)
keypoints_xy = transform.apply_coords(keypoints_xy)

# 得到一个所有的应用的transform的列表
transforms = T.TransformList([transform]).transforms
# 检查是否翻转了奇数次
do_hflip = sum(isinstance(t, T.HFlipTransform) for t in transforms) % 2 == 1
if do_hflip:
    keypoints_xy = keypoints_xy[flip_indices_mapping]
```

4 训练

根据以前的指南，你现在可以已经有一个自定义的模型和数据加载器了。为了训练，用户通常会有下面两种表现之一：

4.1 自定义训练循环

有了模型和数据加载器，为了写一个训练循环，剩余的其他所有部分都可以在 PyTorch 中找到，同时你也可以自己写一个训练循环。这种风格允许研究者更加清晰地管理并完全控制整个训练过程。一个例子是tools/plain_train_net.py。

用户可以轻松控制任何自定义的训练逻辑。

4.2 trainer 摘要

我们也提供了一个有帮助简化标准训练行为的勾子系统的标准“trainer”摘要。它包括下面两个实例：

- SimpleTrainer为单一损失、单一优化器、单一源训练提供了一个最简化版本的训练循环。其他任务（checkpointing，logging 等等）可以通过勾子系统实现。

- DefaultTrainer是一个在tools/train_net.py和其他脚本中使用的，由 yacs 设置初始化的`SimpleTrainer`。它包括更多用户想要选择加入的标准操作，包括优化器的默认设置、学习率调整、logging、评估和 checkpinging 等。

为了自定义一个`DefaultTrainer`：

1. 对于简单自定义（例如，改变优化器、评测器、学习率调整器和数据加载器等），正如tools/train_net.py，在一个子类中重写它的方法。
2. 对于训练中的额外任务，查看是否支持。作为一个例子，在训练中输出 hello：

```
        class HelloHook(HookBase):
            def after_step(self):
                if self.trainer.iter % 100 == 0:
                    print(f"Hello at iteration {self.trainer.iter}!")

```

3. 使用 trainer+hook 系统意味着存在一些不被支持的非标准行为，尤其是在研究中。出于这个原因，我们有意保持 trainer 和 hook 系统最小化而不是功能强大。如果有任何不能被这个系统实现的行为，从tools/plain_train_net.py开始手动实现自定义训练逻辑是更容易的。

4.3 记录度量

在训练中，detectron2 模型和训练器将度量集中放在EventStorage。你可以通过以下代码来访问它并记录度量：

```
from detectron2.utils.events import get_event_storage

# inside the model:
if self.training:
    value = # compute the value from inputs
    storage = get_event_storage()
    storage.put_scalar("some_accuracy", value)
```

更多细节请查看文档。

度量之后会通过EventWriter写到各种目的地。DefaultTrainer 通过默认配置文件会使能一些`EventWriter`。如何自定义它们请见上文。