



Developing Computer Vision Applications Using Existing Detectron2 Models

This chapter guides you through the steps to develop applications for computer vision tasks using state-of-the-art models in Detectron2. Specifically, this chapter discusses the existing and pre-trained models in Detectron2's Model Zoo and the steps to develop applications for object detection, instance segmentation, key-point detection, semantic segmentation, and panoptic segmentation using these models. Thus, you can quickly develop practical computer vision applications without having to train custom models.

By the end of this chapter, you will know what models are available in the Detectron2 Model Zoo. Furthermore, you will learn how to develop applications for vision tasks using these cutting-edge models from Detectron2. Specifically, this chapter covers the following:

- Introduction to Detectron2's Model Zoo
- Developing an object detection application
- Developing an instance segmentation application
- Developing a key-point detection application
- Developing a panoptic segmentation application
- Developing a semantic segmentation application
- Putting it all together

Technical requirements

You should have set up the development environment with the instructions provided in [Chapter 1](#). Thus, if you have not done so, please complete setting up the development environment before continuing. All the code, datasets, and respective results are available on the GitHub page of the book (under the folder named `chapter02`) at <https://github.com/PacktPublishing/Hands-On-Computer-Vision/tree/master/Chapter02>

[tPublishing/Hands-On-Computer-Vision-with-Detectron2](#). It is highly recommended to download the code and follow along.

Introduction to Detectron2's Model Zoo

In deep learning, when developing large models and training models on massive datasets, developers of the deep learning methods often provide pre-trained models. The main reason for this is that the developers of these models usually are the big players in the field (e.g., Facebook, Google, Microsoft, or universities) with access to computation resources (e.g., CPUs and GPUs) to train such models on large datasets. These computation resources are generally not accessible for standard developers elsewhere.

These pre-trained models are often trained on the datasets for uses with common tasks that the models are intended for. Therefore, another benefit of such models is for the users to adopt them and use them for specific cases of the tasks at hand if they match what the models were trained for. Furthermore, these models can also be used as the baselines on which the end users can further fine-tune them for their specific cases with smaller datasets that require fewer computation resources to train.

Detectron2 also provides an extensive collection of models trained with Big Basin servers with high computation resources (8 NVIDIA V100 GPUs and NVLink) on its Model Zoo. You can read further information and download all the models from this link: https://github.com/facebookresearch/detectron2/blob/main/MODEL_ZOO.md.

It would be a little overwhelming for new users to grasp information about the available models with different specifications. So, this section details which information users should look for to complete a specific computer vision task at hand using these cutting-edge models from Detectron2 Model Zoo. These items include finding information about the following:

- The dataset on which the model was trained
- The architecture of this model
- Training and inferencing times

- Training memory requirement
- Model accuracy
- The link to the trained model (weights)
- The link to its configuration file (`.yaml`)

Knowing the dataset on which the model was trained gives information about the available tasks that this specific model can offer. For instance, a dataset with class labels and bounding boxes only provides models for object detection tasks. Similarly, datasets that have class labels, bounding boxes, and pixel-level annotations per individual object offer models that provide object detection and instance segmentation, and so on.

Furthermore, the specific class labels from the datasets are also significant. For instance, if the computer vision task is detecting humans from videos, and the dataset on which the model was trained does not have a class label for humans (people), then such a model cannot be used for this specific task directly.

The two most common image datasets for computer vision tasks are ImageNet and **COCO (Common Objects in Context)**. ImageNet currently has more than 14 million images (14,197,122 images by this writing). Out of these 14 million images, more than 1 million have bounding boxes for the dominant object in the image. Note also that when the computer vision community refers to ImageNet, they refer to a subset of these images curated by **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)**. This subset has 1,000 object categories and more than 1.2 million images. More than half of these images are annotated with bounding boxes around the objects of the image category. This dataset is often used to train cutting-edge deep learning models such as VGG, Inception, ResNet, and RestNeXt. The number of classes in this dataset is enormous. Therefore, your computer vision task at hand would likely fall to detecting one or more of the 1,000 specified categories. Thus, the chance of reusability of the models trained on this dataset is high.

Microsoft is also another major player in this area. Specifically, Microsoft also publishes the COCO dataset. This dataset contains (at the time of writing this book) 330,000 images, with more than 200,000 images that are labeled. There are about 1.5 million object instances with 80 object categories and 91 stuff categories. Notably, there are also 250,000 people with keypoints to support training models for keypoint detection tasks. All Detectron2's pre-trained COCO models were trained and evaluated on the

COCO 2017 dataset. The train, validation, and test sets of COCO 2017 can be downloaded using the following commands:

```
$ wget http://images.cocodataset.org/zips/train2017.zip
$ wget http://images.cocodataset.org/zips/val2017.zip
$ wget http://images.cocodataset.org/zips/test2017.zip
```

IMPORTANT NOTE

ImageNet and COCO datasets are being used and frequently updated by the computer vision community. Interested readers can refer to the official web pages of these datasets for the latest updates and computer vision challenges related to them. These web pages are <https://image-net.org> and <https://cocodataset.org>, respectively.

The next important piece of information related to any pre-trained model is the architecture of the pre-trained model. This is often reflected by the model's name in the Model Zoo. For instance, **R50** and **R101** mean **Microsoft Research Asia (MSRA) ResNet-50** and **ResNet-101**, respectively. Similarly, **x101** and **x152** mean **ResNeXt-101-32x8d** and **ResNeXt-152-32x8d**, respectively. Some model names include **FPN**. This part of the name indicates that the model uses the **Feature Pyramid Network (FPN)** technique. This technique means that the features are extracted at different layers of the model's architecture (instead of taking the output feature from the last layer and feed to the prediction process). Using the FPN backbone helps to increase the accuracy with the time trade-off. Furthermore, having **c4** in its name means that this model uses the **conv4** backbone. Similarly, having **dc5** in its name indicates that this model uses **conv5** with dilations in **conv5**.

IMPORTANT NOTE

Interested readers can refer to the web pages regarding the ResNet architecture at <https://github.com/KaimingHe/deep-residual-networks> and the ResNeXt architecture at <https://github.com/facebookresearch/ResNeXt> for further information.

After knowing the trained dataset and the model architecture, the next step is to check for the training and inferencing time, training memory, and model accuracy. This information helps to decide the hardware re-

quirements and the trade-off between speed and accuracy. This information is also listed for every pre-trained model on the Detectron2 Model Zoo.

After deciding which model to work with based on speed versus hardware requirement trade-off, the next step is downloading the model's configuration file. On the Model Zoo, the model's name is linked with a `.yaml` configuration file. **YAML** (short for **Yet Another Markup Language**, or **YAML ain't markup language**) is a human-readable programming language for serializing data. It is commonly used for writing configuration files and supporting automation processes. Finally, there is also a link to the weights of the pre-trained model for downloading.

By this time, you should be able to read and understand important information about the pre-trained models available on the Detectron2 Model Zoo. The next step is using some pre-trained models to perform important computer vision tasks. Developing computer vision applications using Detectron2 typically includes the following steps:

- Selecting a configuration file
- Getting a predictor
- Performing inferences
- Visualizing results

The following sections detail these steps for developing object detection applications.

Developing an object detection application

Object detection generally includes object localization and classification. Specifically, deep learning models for this task predict where objects of interest are in an image by giving the bounding boxes around these objects (localization). The following sections detail the steps to develop an object detection application using Detectron2 pre-trained models.

Getting the configuration file

Various pre-trained models for object detection tasks are available on the Detectron2 Model Zoo. These models are listed under the **COCO Object Detection Baselines** header. Specifically, there are three main categories of these baselines, namely the Faster **Region-based Convolution Neural Network (R-CNN)** subheader, the RetinaNet subheader, and the **Region Proposal Network (RPN)** and Fast R-CNN subheader. *Figure 2.1* shows the available pre-trained models currently listed under the Faster R-CNN subheader.

Faster R-CNN:

Name	lr sched	train time (s/iter)	inference time (s/im)	train mem (GB)	box AP	model id	download
R50-C4	1x	0.551	0.102	4.8	35.7	137257644	model metrics
R50-DC5	1x	0.380	0.068	5.0	37.3	137847829	model metrics
R50-FPN	1x	0.210	0.038	3.0	37.9	137257794	model metrics
R50-C4	3x	0.543	0.104	4.8	38.4	137849393	model metrics
R50-DC5	3x	0.378	0.070	5.0	39.0	137849425	model metrics
R50-FPN	3x	0.209	0.038	3.0	40.2	137849458	model metrics
R101-C4	3x	0.619	0.139	5.9	41.1	138204752	model metrics
R101-DC5	3x	0.452	0.086	6.1	40.6	138204841	model metrics
R101-FPN	3x	0.286	0.051	4.1	42.0	137851257	model metrics
X101-FPN	3x	0.638	0.098	6.7	43.0	139173657	model metrics

Figure 2.1: Pre-trained Faster R-CNN models for object detection

This list details the model name (which reflects its architecture), train and inference times, memory requirement, accuracy (box **AP** or **average precision**), and links to download the model configuration file and weights.

For demonstration purposes, the application described in this section selects the **x101-FPN** pre-trained model listed at the end of the list for the object detection task. The model's name indicates that it is the **ResNeXt-101-32x8d-FPN** network architecture. This name is linked with the model's configuration file (the **.yaml** file), and the following is the link for this specific model:

https://github.com/facebookresearch/detectron2/blob/main/configs/COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml

Similarly, there is a link to the pre-trained weights of the model (under the **download** column and the **model** links in *Figure 2.1*). These links can

be used to download the corresponding configuration and trained weights for this model. However, this model is currently distributed together with Detectron2's installation. For instance, after installing Detectron2 on Google Colab, you can execute the following commands on a code cell to find out what models are released together with the installation (please note the python version as you may need to change it to reflect the current Python version used):

```
!sudo apt-get install tree
!tree /usr/local/lib/python3.7/dist-packages/detectron2/model_zoo
```

Figure 2.2 shows a part of the resulting output of this code snippet. The output indicates which models are available for **COCO-Detection** (models for object detection trained on the COCO dataset).

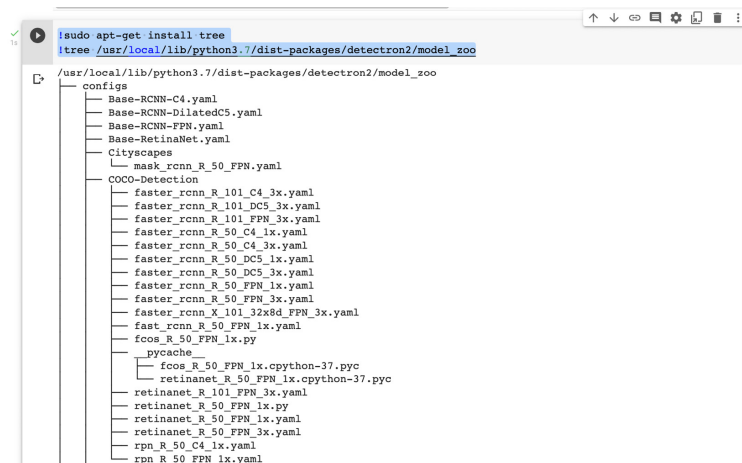


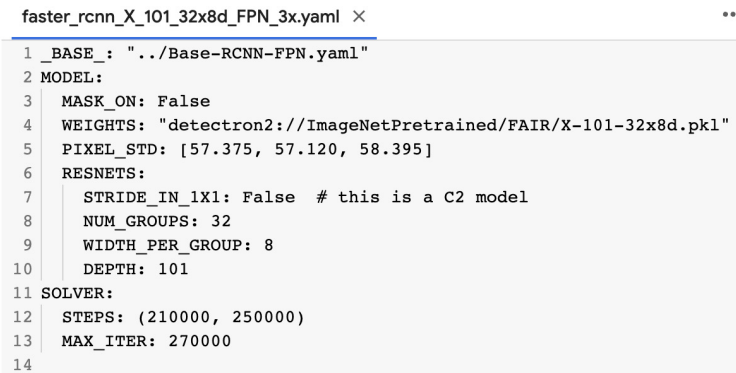
Figure 2.2: Configuration files for models trained on the COCO dataset for object detection

Therefore, only the relative path to this configuration file is necessary. In other words, the configuration file can be accessed from the **configs** folder in the **model_zoo** folder in Detectron2's installation folder. The following is the relative path to the selected configuration file:

```
COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml
```

Notably, this is the part at the end of the complete link listed previously after the **/configs/** folder. By reading this configuration file, Detectron2 knows what model configurations are and where to download the model

weights. *Figure 2.3* shows the configuration file for the model selected in this section.



```
faster_rcnn_X_101_32x8d_FPN_3x.yaml × ***
1 _BASE_: "../Base-RCNN-FPN.yaml"
2 MODEL:
3   MASK_ON: False
4   WEIGHTS: "detectron2://ImageNetPretrained/FAIR/X-101-32x8d.pkl"
5   PIXEL_STD: [57.375, 57.120, 58.395]
6   RESNETS:
7     STRIDE_IN_1X1: False # this is a C2 model
8     NUM_GROUPS: 32
9     WIDTH_PER_GROUP: 8
10    DEPTH: 101
11 SOLVER:
12   STEPS: (210000, 250000)
13   MAX_ITER: 270000
14
```

Figure 2.3: Configuration file for the ResNeXt101 pre-trained model

We now know how to get the configuration file for the intended model. This configuration file also gives information for Detectron2 to download its pre-trained weights. We are ready to use this configuration file and develop the object detection application.

Getting a predictor

The following code snippet imports the required libraries and creates a model (a predictor interchangeably):

```
import detectron2
from detectron2.config import get_cfg
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
# Suppress some user warnings
import warnings
warnings.simplefilter(action='ignore', category=UserWarning)
# Select a model
config_file = "COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml"
checkpoint_url = "COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml"
# Create a configuration file
cfg = get_cfg()
config_file = model_zoo.get_config_file(config_file)
cfg.merge_from_file(config_file)
# Download weights
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url(checkpoint_url)
```



```
score_thresh_test = 0.95
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = score_thresh_test
predictor = DefaultPredictor(cfg)
```

Specifically, the `get_cfg` method helps to get a default configuration object for this model. Similarly, `model_zoo` helps to get the configuration file (the `.yaml` file) and download the pre-trained weights. At the same time, `DefaultPredictor` is required to create the predictor from the downloaded configuration file and weights. Besides the required imports, this code snippet also suppresses several user warnings that are safe to ignore.

Notably, the `cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST` parameter is currently set to `0.95`. This parameter is used at the test time to filter out all the detected objects with detection confidences less than this value. Assigning this parameter to a high value (close to 1.0) means we want to have high precision and low recall. Likewise, setting it to a low value (close to 0.0) means we prefer low precision and high recall. Once we have the model (or predictor interchangeably), it is time to perform inferences.

Performing inferences

The following code snippet performs object detection on an input image. Specifically, we first import `cv2` to read an image from an `input_url`. Next, using the predictor created earlier, it is relatively easy to perform object detection on the loaded image (please upload an image to Google Colab to perform inference):

```
import cv2
input_url = "input.jpeg"
img = cv2.imread(input_url)
output = predictor(img)
```

The next important task is understanding what output format Detectron2 gives and what each output component means. This `print(output)` displays the `output` variable and the fields it provides:

```
{'instances': Instances(num_instances = 4, image_height = 720, image_width = 960, fields = [pred_boxes: Boxes(tensor([[492
[294.6652, 192.1288, 445.8517, 655.3390],
[419.8898, 253.9013, 507.7427, 619.1491],
```

```
[451.8833, 395.1392, 558.5139, 671.0604]], device = 'cuda:0')), scores: tensor([0.9991, 0.9986, 0.9945, 0.9889], device=
)
```

Observably, besides the image information, the object detection result also includes the detected instances (**instances**) with corresponding predicted bounding boxes (**pred_boxes**), predicted confidence scores (**scores**), and the predicted class labels (**pred_classes**). With this information, we are ready to visualize the detected objects with corresponding bounding boxes.

Visualizing the results

Detectron2 provides a **visualizer** class, which helps you to easily annotate detected results to the input image. Thus, the following code snippet imports this class and uses its visualization method to annotate detection results to the input image:

```
from google.colab.patches import cv2_imshow
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog
metadata = MetadataCatalog.get(cfg.DATASETS.TRAIN[0])
v = Visualizer(img[:, :, ::-1], metadata, scale=0.5)
instances = output["instances"].to("cpu")
annotated_img = v.draw_instance_predictions(instances)
cv2_imshow(annotated_img.get_image()[:, :, ::-1])
```

Specifically, the first statement imports the **cv2_imshow** method, a Google Colab patch, to display images to Google Colab output. The next **import** statement is **visualizer** itself. The third **import** statement helps get information about the training dataset. Specifically, this **MetadataCatalog** object provides the label names for the predicted objects. In this case, a class label is a **person** for the predicted class number 0, as in the output.

The **visualizer** class then uses the original image data and the metadata to create a **Visualizer** object. Notably, this **Visualizer** object uses a different image format than OpenCV (**cv2**). Therefore, we need to use the slice operator as **[:, :, ::-1]** to convert the image format between the two. Finally, we call **draw_instance_predictions** from the visualizer object to annotate the predicted instances to the image and use the **cv2_imshow** method to display the result to the output. *Figure 2.4* shows an example output image after using the **Visualizer** object to annotate the inference results to the corresponding input image.

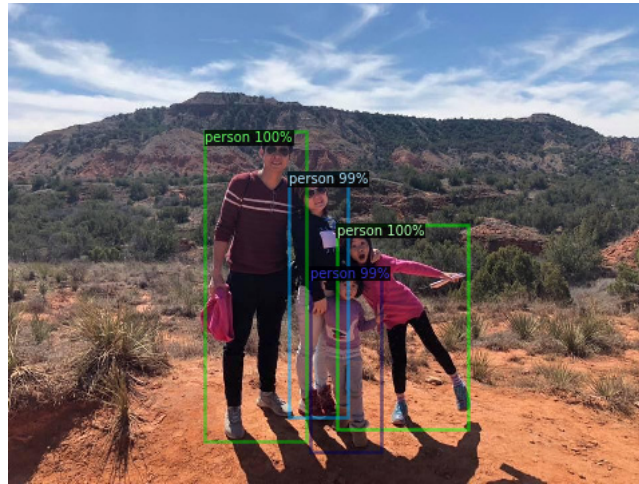


Figure 2.4: Inference output with object detection results annotated

Congratulations! You have completed the first object detection application using Detectron2. Next, we will move on to develop an instance segmentation application using Detectron2.

Developing an instance segmentation application

Like object detection, instance segmentation also involves object localization and classification. However, instance segmentation takes one step further while localizing the detected objects of interest. Specifically, besides classification, models for this task localize the detected objects at the pixel level. The following sections detail the steps to develop an instance segmentation application using Detectron2 pre-trained models.

Selecting a configuration file

Like object detection, Detectron2 also provides a list of cutting-edge models pre-trained for object instance segmentation tasks. For instance, *Figure 2.5* shows the list of Mask R-CNN models pre-trained on the *COCO Instance Segmentation* dataset.

Name	lr sched	train time (s/iter)	inference time (s/im)	train mem (GB)	box AP	mask AP	model id	download
R50-C4	1x	0.584	0.110	5.2	36.8	32.2	137259246	model metrics
R50-DC5	1x	0.471	0.076	6.5	38.3	34.2	137260150	model metrics
R50-FPN	1x	0.261	0.043	3.4	38.6	35.2	137260431	model metrics
R50-C4	3x	0.575	0.111	5.2	39.8	34.4	137849525	model metrics
R50-DC5	3x	0.470	0.076	6.5	40.0	35.9	137849551	model metrics
R50-FPN	3x	0.261	0.043	3.4	41.0	37.2	137849600	model metrics
R101-C4	3x	0.652	0.145	6.3	42.6	36.7	138363239	model metrics
R101-DC5	3x	0.545	0.092	7.6	41.9	37.3	138363294	model metrics
R101-FPN	3x	0.340	0.056	4.6	42.9	38.6	138205316	model metrics
X101-FPN	3x	0.690	0.103	7.2	44.3	39.5	139653917	model metrics

Figure 2.5: COCO Instance Segmentation baselines with Mask R-CNN

After checking the specifications of these models, this specific application selects the **x101-FPN** model. The following is the relative path to this configuration file (extracted and cut out from the address linked with the model name):

```
COCO-InstanceSegmentation/mask_rcnn_X_101_32x8d_FPN_3x.yaml
```

Getting a predictor

The code snippet for getting a predictor for this object instance segmentation remains the same for the previously developed object detection application except for the following two lines. These two lines specify different configuration files for this specific pre-trained model:

```
# Select a model
config_file = "COCO-InstanceSegmentation/mask_rcnn_X_101_32x8d_FPN_3x.yaml"
checkpoint_url = "COCO-InstanceSegmentation/mask_rcnn_X_101_32x8d_FPN_3x.yaml"
```

This simple modification is an example of an excellent design of Detectron2. It is relatively easy to switch from one type of application to another by just simply modifying the link to the configuration files.

Performing inferences

The inferencing code snippet remains the same as the object detection application developed in the previous section. However, the object instance

segmentation output has some differences. Specifically, besides `pred_boxes`, `scores`, and `pred_classes`, for the bounding boxes, prediction confidence scores, and corresponding predicted class labels, there is also a tensor (`pred_masks`) to indicate whether an individual pixel from the input picture belongs to a detected instance or not (`True/False`). The following is an example of this tensor (a part of the output was cut off for space efficiency) for the four detected instances:

```
pred_masks: tensor([
  [[False, False, False, ..., False, False, False],
   ...,
   [False, False, False, ..., False, False, False]],
  [[False, False, False, ..., False, False, False],
   ...,
   [False, False, False, ..., False, False, False]],
  [[False, False, False, ..., False, False, False],
   ...,
   [False, False, False, ..., False, False, False]],
  [[False, False, False, ..., False, False, False]]],
  device='cuda:0')
```

Visualizing the results

Thanks to the excellent design of Detectron2, visualizing instance segmentation results is the same as visualizing object detection results. In other words, the code snippet remains the same. *Figure 2.6* shows the input image annotated with instance segmentation results. Specifically, besides the bounding boxes and corresponding prediction confidences, pixels belonging to each detected individual are also highlighted.

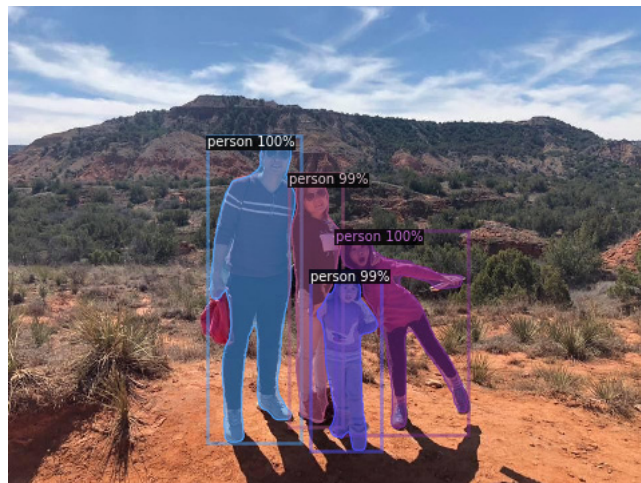


Figure 2.6: Inferencing output with object instance segmentation results

Congratulations! You now completed an object instance segmentation application using a Detectron2 cutting-edge model with only a few modifications compared to the previously developed object detection application. In the next section, you will learn how to develop a keypoint detection application using Detectron2.

Developing a keypoint detection application

Besides detecting objects, keypoint detection also indicates important parts of the detected objects called **keypoints**. These keypoints describe the detected object's essential trait. This trait is often invariant to image rotation, shrinkage, translation, or distortion. The following sections detail the steps to develop a keypoint detection application using Detectron2 pre-trained models.

Selecting a configuration file

Detectron2 also provides a list of cutting-edge algorithms pre-trained for keypoint detection for human objects. For instance, *Figure 2.7* shows the list of Mask R-CNN models pre-trained on the *COCO Person Keypoint Detection* dataset.

Name	lr sched	train time (s/iter)	inference time (s/im)	train mem (GB)	box AP	kp. AP	model id	download
R50-FPN	1x	0.315	0.072	5.0	53.6	64.0	137261548	model metrics
R50-FPN	3x	0.316	0.066	5.0	55.4	65.5	137849621	model metrics
R101-FPN	3x	0.390	0.076	6.1	56.4	66.1	138363331	model metrics
X101-FPN	3x	0.738	0.121	8.7	57.3	66.0	139686956	model metrics

Figure 2.7: COCO Person Keypoint Detection baselines with Keypoint R-CNN

In this specific case, we select the **x101-FPN** pre-trained model. Again, the link to the configuration file is linked with the model name in the first column, and we only use the part after **/configs/**. Thus, here is the link to the configuration file for this model:

COCO-Keypoints/keypoint_rcnn_X_101_32x8d_FPN_3x.yaml

Getting a predictor

The code for getting the model remains the same, except for the lines that set the links to the model configuration and weights:

```
# Select a model
config_file = "COCO-Keypoints/keypoint_rcnn_X_101_32x8d_FPN_3x.yaml"
checkpoint_url = "COCO-Keypoints/keypoint_rcnn_X_101_32x8d_FPN_3x.yaml"
```

Performing inferences

The code snippet for performing inferences on an input image remains the same. However, the output is a little different this time. Specifically, besides the image information, **pred_boxes**, **scores**, and **pred_classes**, there is a tensor that contains information about the predicted keypoints (**pred_keypoints**). There is also another field (**pred_keypoint_heatmaps**) for storing the keypoint heatmaps of the detected person objects. Due to space limitations, this output is not listed here. Please see this application's notebook on the book's GitHub repository, as mentioned in the *Technical requirements* section to view the sample output.

Visualizing the results

Once again, thanks to the flexible design of Detectron2. The visualization code for the keypoint detection application remains the same for object detection and object instance segmentation applications. *Figure 2.8* shows the input image annotated with detected keypoints for detected people.

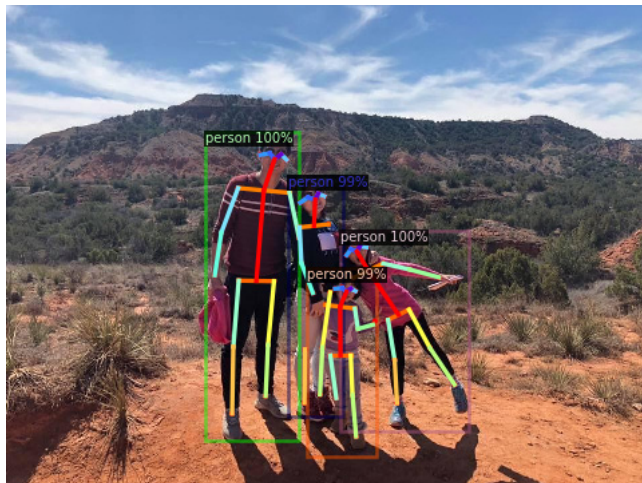


Figure 2.8: Inference output with person keypoint detection results

Congratulations! With only two minor modifications to the object instance segment code, you can now create an application to detect person keypoints using Detectron2. In the next section, you will learn how to develop another computer vision application for panoptic segmentation tasks.

Developing a panoptic segmentation application

Panoptic literally means “everything visible in the image.” In other words, it can be viewed as a combination of common computer vision tasks such as instance segmentation and semantic segmentation. It helps to show the unified and global view of segmentation. Generally, it classifies objects in an image into foreground objects (that have proper geometries) and background objects (that do not have appropriate geometries but are textures or materials). The following sections detail the steps to develop a panoptic segmentation application using Detectron2 pre-trained models.

Selecting a configuration file

Detectron2 provides several pre-trained models for panoptic segmentation tasks. For instance, *Figure 2.9* shows the list of panoptic segmentation models trained on the COCO Panoptic Segmentation dataset.

Name	lr sched	train time (s/iter)	inference time (s/im)	train mem (GB)	box AP	mask AP	PQ	model id	download
R50-FPN	1x	0.304	0.053	4.8	37.6	34.7	39.4	139514544	model metrics
R50-FPN	3x	0.302	0.053	4.8	40.0	36.5	41.5	139514569	model metrics
R101-FPN	3x	0.392	0.066	6.0	42.4	38.5	43.0	139514519	model metrics

Figure 2.9: COCO Panoptic Segmentation baselines with panoptic FPN

This specific application in this section selects the **R101-FPN** model to perform panoptic segmentation tasks. The path to the configuration file of this model is the following:

```
COCO-PanopticSegmentation/panoptic_fpn_R_101_3x.yaml
```

Getting a predictor

Similar to the previous applications, the code snippet for panoptic segmentation remains the same, except for the lines that set the configuration and the weights:

```
# Select a model
config_file = "COCO-PanopticSegmentation/panoptic_fpn_R_101_3x.yaml"
checkpoint_url = "COCO-PanopticSegmentation/panoptic_fpn_R_101_3x.yaml"
```

Performing inferences

Once again, the code snippet for performing inferences remains the same. However, the output has some other fields for the panoptic segmentation tasks. Specifically, it contains a field called **panoptic_seg** for the predicted panoptic segmentation information. The following is the sample output of this field:

```
'panoptic_seg': (tensor([[6, 6, 6, ..., 6, 6, 6],
[6, 6, 6, ..., 6, 6, 6],
[6, 6, 6, ..., 6, 6, 6],
...,
```

```
[9, 9, 9, ..., 9, 9, 9],
[9, 9, 9, ..., 9, 9, 9],
[9, 9, 9, ..., 9, 9, 9]], device = 'cuda:0', dtype = torch.int32),
[{'id': 1, 'isthing': True, 'score': 0.9987247586250305, 'category_id': 0, 'instance_id': 0 },
{'id': 2, 'isthing': True, 'score': 0.997495174407959, 'category_id': 0, 'instance_id': 1 },
{'id': 3, 'isthing': True, 'score': 0.9887707233428955, 'category_id': 0, 'instance_id': 2 },
{'id': 4, 'isthing': True, 'score': 0.9777324199676514, 'category_id': 0, 'instance_id': 3 },
{'id': 5, 'isthing': False, 'category_id': 37, 'area': 15173 },
{'id': 6, 'isthing': False, 'category_id': 40, 'area': 163644 },
{'id': 7, 'isthing': False, 'category_id': 45, 'area': 192040 },
{'id': 8, 'isthing': False, 'category_id': 46, 'area': 62182 },
{'id': 9, 'isthing': False, 'category_id': 47, 'area': 164592 }])
```

Specifically, this field is a tuple with two elements. The first element is a tensor of the same size as the input image. This tensor helps to indicate the detected class label number for every pixel from the input image. The second element of this tuple is a list of items that map corresponding label numbers into known classes.

Visualizing the results

The visualization code snippet for panoptic segmentation is very similar to other vision tasks we have learned, except that we need to extract the two elements of the `panoptic_seg` field (described previously) from the output. Next, these two fields are passed to the `draw_panoptic_seg_predictions` method of the `Visualizer` object to perform the annotations:

```
from google.colab.patches import cv2_imshow
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog
metadata = MetadataCatalog.get(cfg.DATASETS.TRAIN[0])
v = Visualizer(img[:, :, ::-1], metadata, scale=0.5)
panoptic_seg, segments_info = output["panoptic_seg"]
annotated_img = v.draw_panoptic_seg_predictions(panoptic_seg.to("cpu"), segments_info)
cv2_imshow(annotated_img.get_image()[:, :, ::-1])
```

Figure 2.10 shows the panoptic segmentation results for an input image. Specifically, every pixel in the input image is classified as either one of the four detected people, the sky, a mountain, grass, a tree, or dirt (a total of nine label values).

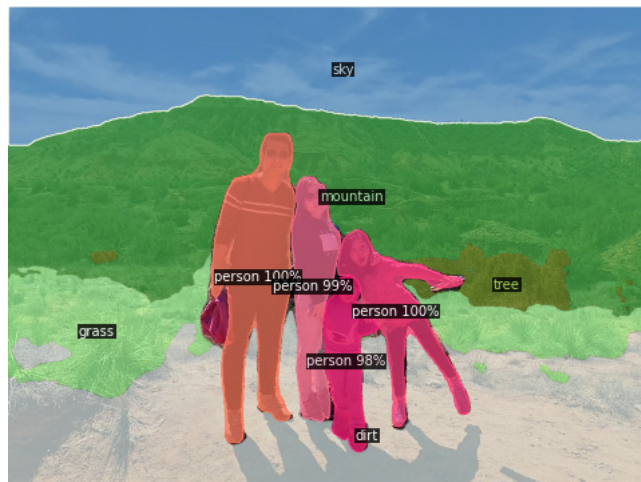


Figure 2.10: Inferencing output with person keypoint detection results

Congratulations! You now know how to develop a cutting-edge panoptic segmentation application. Though there are a few changes for this panoptic segmentation application compared to the previous applications, the main code is pretty much the same. In the next section, you will learn how to develop an application for semantic segmentation tasks.

Developing a semantic segmentation application

A semantic segmentation task does not detect specific instances of objects but classifies each pixel in an image into some classes of interest. For instance, a model for this task classifies regions of images into pedestrians, roads, cars, trees, buildings, and the sky in a self-driving car application. The following sections detail the steps to develop a semantic segmentation application using Detectron2 pre-trained models.

Selecting a configuration file and getting a predictor

Semantic segmentation is a byproduct of panoptic segmentation. For instance, it groups detected objects of the same class into one if they are in the same region instead of providing segmentation data for every detected object. Therefore, the model for semantic segmentation is the same as

that for panoptic segmentation. Therefore, the configuration file, the weights, and the code snippet for getting a predictor are the same as those for the previous panoptic segmentation application.

Performing inferences

The inferencing code snippet for semantic segmentation is also the same for panoptic segmentation. The inferencing result (the same as that for panoptic segmentation) also has a field called `sem_seg`. This field has the shape of `classes × image_height × image_width`. Specifically, it is `54 × 720 × 960` in this specific case. The size `720 × 960` corresponds to all pixels of the input image. For each pixel, there are 54 values, corresponding to the probabilities of the pixel being classified into 54 categories of the trained dataset, respectively.

Visualizing the results

The visualization of the semantic segmentation results is slightly different. First, the `sem_seg` field is extracted. Next, from the 54 probabilities, the class label with the highest predicted probability is selected as the label for every individual pixel using the `argmax` method. The extracted data is then passed to the `draw_sem_seg` method from the `Visualizer` object to perform the visualization. Here is the code snippet for visualizing the semantic segmentation result:

```
from google.colab.patches import cv2_imshow
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog
metadata = MetadataCatalog.get(cfg.DATASETS.TRAIN[0])
v = Visualizer(img[:, :, ::-1], metadata, scale=0.5)
sem_seg = output["sem_seg"].argmax(dim=0)
annotated_img = v.draw_sem_seg(sem_seg.to("cpu"))
cv2_imshow(annotated_img.get_image()[:, :, ::-1])
```

Figure 2.11 shows the semantic segmentation result visualized on the corresponding input image. Notably, compared to the panoptic results, the detected people are grouped into one region and classified as `things`.

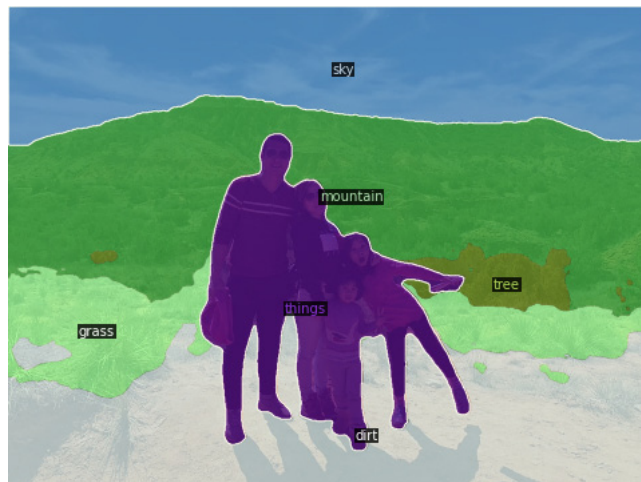


Figure 2.11 – Inference output with semantic segmentation results

Congratulations! You now mastered the steps to create computer vision applications to perform important computer vision tasks using cutting-edge models available on Detectron2. Thanks to Detectron2's flexible design, the code snippets for these applications are very similar to one another. Therefore, the next section puts all these together and develops an application that can perform all these tasks.

Putting it all together

After developing different applications for different computer vision tasks in the previous sections, it is clear that Detectron2 is highly structured because the described applications have the same structure. Therefore, it is reasonable to implement these steps to perform all these tasks. The following sections cover the steps to develop a computer vision application that can perform different computer vision tasks using Detectron2 pre-trained models.

Getting a predictor

The code snippets for getting a predictor for different computer vision tasks are similar. They all need to perform some basic imports, take the configuration file, weight file, and score threshold at test time. Therefore, we create a function called `get_predictor` for this. Here is the code snippet for this function:

```

import detectron2
from detectron2.config import get_cfg
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
# We suppress some user warnings
import warnings
warnings.simplefilter(action='ignore',category=UserWarning)
# Create a configuration and a predictor
def get_predictor(config_file, checkpoint_url, score_thresh_test=1.0):
    # create a configuration object
    cfg = get_cfg()
    # get the configurations from the config_file
    config_file = model_zoo.get_config_file(config_file)
    cfg.merge_from_file(config_file)
    # get the pre-built weights of a trained model from the checkpoint
    cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url(checkpoint_url)
    # set the threshold for recall vs. precision at test time
    cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = score_thresh_test
    # create a predictor
    predictor = DefaultPredictor(cfg)
    return cfg, predictor

```

To get a model, you only need to select the paths to the configuration file and pre-trained weights and pass them to this method. This method then returns a predictor (**predictor**) together with this model's configuration (**cfg**). After having the predictor, the next step is to perform inferences.

Performing inferences

The code snippet for all the computer vision tasks is precisely the same. Therefore, we should provide a method for this (**perform_inference**). Here is the code snippet for performing inference given any predictor (**predictor**) and the path to the input image (**input_url**):

```

import cv2
from google.colab.patches import cv2_imshow
def perform_inference(predictor, input_url):
    img = cv2.imread(input_url)
    output = predictor(img)
    return img, output

```

This method (**perform_inference**) returns the input image and corresponding output predicted by the predictor on the input image. The next step is to have a method to annotate this output to the input image.

Visualizing the results

The code snippets for visualizing object detection, object instance segmentation, and keypoint detection are the same. However, they are slightly different if the task is panoptic or semantic segmentations. Therefore, there are constants to indicate which task to be performed, and the visualization method (called `visualize_output`) can perform appropriate visualization based on the input task. Here is the code snippet for the result visualizations:

```
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog
# computer vision tasks
OBJECT_DETECTION = 0
INSTANCE_SEGMENTATION = 1
KEYPOINT_DETECTION = 2
SEMANTIC_SEGMENTATION = 3
PANOPTIC_SEGMENTATION = 4
def visualize_output(img, output, cfg, task=OBJECT_DETECTION, scale=1.0):
    v = Visualizer(img[:, :, ::-1], MetadataCatalog.get(cfg.DATASETS.TRAIN[0]), scale=scale)
    if task == PANOPTIC_SEGMENTATION:
        panoptic_seg, segments_info = output["panoptic_seg"]
        annotated_img = v.draw_panoptic_seg_predictions(panoptic_seg.to("cpu"), segments_info)
    elif task == SEMANTIC_SEGMENTATION:
        sem_seg = output["sem_seg"].argmax(dim=0)
        annotated_img = v.draw_sem_seg(sem_seg.to("cpu"))
    else:
        annotated_img = v.draw_instance_predictions(output["instances"].to("cpu"))
    cv2_imshow(annotated_img.get_image()[:, :, ::-1])
```

Performing a computer vision task

After having all the corresponding methods for getting a predictor (`get_predictor`), performing inference (`perform_inference`), and visualizing output (`visualize_output`), we are ready to perform detections. Here is the code snippet for performing object detection tasks using these methods:

```
config_file = "COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml"
checkpoint_url = "COCO-Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml"
score_thresh_test = 0.95
input_url = "input.jpeg"
task = OBJECT_DETECTION
```

```
scale = 0.5
# get a predictor
cfg, predictor = get_predictor(config_file, checkpoint_url, score_thresh_test)
# perform inference
img, output = perform_inference(predictor, input_url)
# visualization
visualize_output(img, output, cfg, task, scale)
```

Other computer vision tasks can be performed in the same way.

Furthermore, these methods can be placed into a library (a `*.py` file), so you can include them and use them in any computer vision applications if appropriate.

Summary

This chapter discussed the pre-trained models available on the Detectron2 Model Zoo. Specifically, it specified the vital information to look for while selecting a pre-trained model for a computer vision task. It then provided the code snippets for developing an object detection application, an object instance segmentation application, a keypoint detection application, a panoptic segmentation application, and a semantic segmentation application. The code snippets for these applications are similar. Therefore, the code snippets are abstracted into typical methods for getting a model, performing inferences, and visualizing outputs. These methods can then be reused to develop different computer vision applications with Detectron2 quickly.

The cutting-edge models on Detectron2 are trained with many images and a vast amount of computation resources. Therefore, these models have high accuracies. Additionally, they are also trained on the datasets with class labels for everyday computer vision tasks. Thus, there is a high chance that they can meet your individual needs. However, if Detectron2 does not have a model that meets your requirement, you can train models on custom datasets using Detectron2. *Part 2* of this book covers the steps to prepare data, train, and fine-tune a Detectron2 model on a custom dataset for the object detection task.