Detectron 2

Detectron 2 [2] is a next-generation open-source object detection system from Facebook AI Research. With the repo you can use and train the various state-of-the-art models for detection tasks such as bounding-box detection, instance and semantic segmentation, and person keypoint detection.

https://github.com/facebookresearch/detectron2

Faster R-CNN FPN architecture:

As an example I choose the Base (Faster) R-CNN with Feature Pyramid Network[3] (Base-RCNN-FPN), which is the basic bounding box detector extendable to Mask R-CNN[4]. Faster R-CNN[5] detector with FPN backbone is a multi-scale detector that realizes high accuracy for detecting tiny to large objects, making itself the de-facto standard detector.

Let's look at the structure of the Base R-CNN FPN:

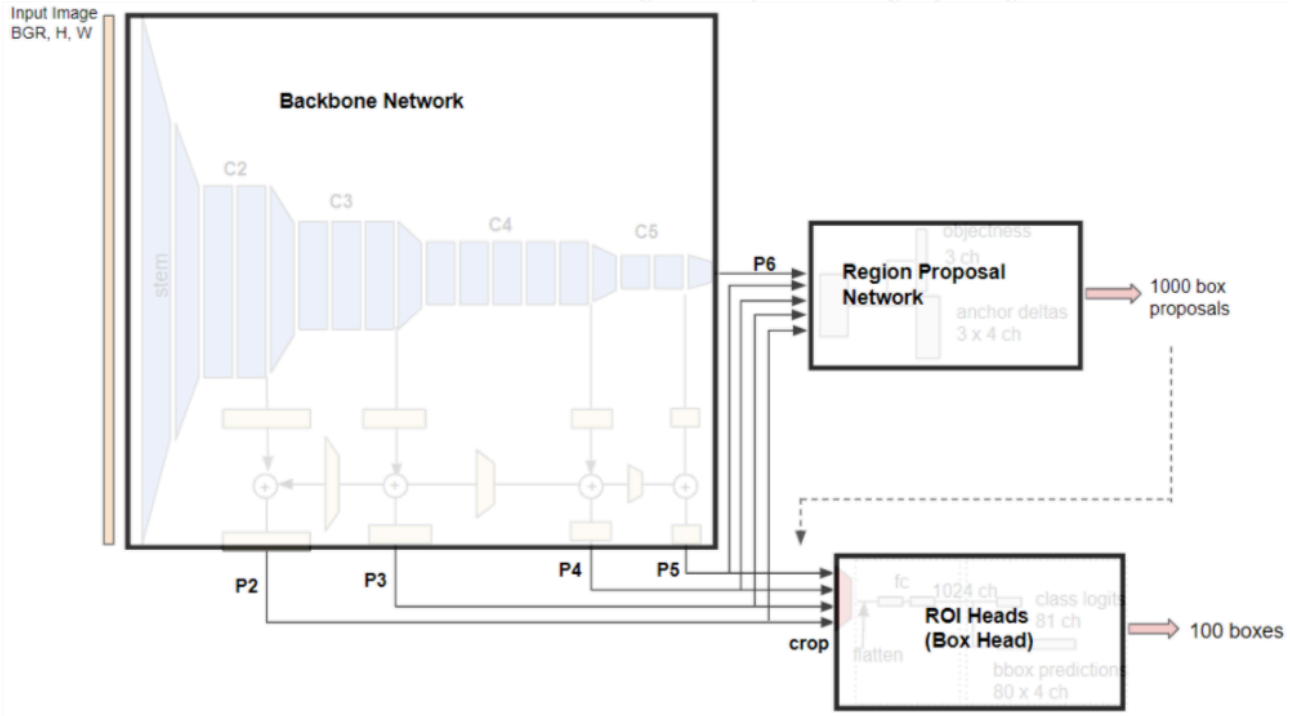


Fig 1: Meta Architecture of Base RCNN FPN

The schematic above shows the meta architecture of the network. Now you can see there are *three blocks* in it, namely:

1. **Backbone Network**: extracts feature maps from the input image at different scales. Base-RCNN-FPN's output features are called P2 (1/4 scale), P3 (1/8), P4 (1/16), P5 (1/32) and P6 (1/64). Note that non-FPN ('C4') architecture's output feature is only from the 1/16 scale.

2. **Region Proposal Network**: detects object regions from the multi-scale features. 1000 box proposals (by default) with confidence scores are obtained.
3. **Box Head**: crops and warps feature maps using proposal boxes into multiple fixed-size features, and obtains fine-tuned box locations and classification results via fully-connected layers. Finally 100 boxes (by default) in maximum are filtered out using non-maximum suppression (NMS). The box head is one of the sub-classes of **ROI Heads**. For example Mask R-CNN has more ROI heads such as a mask head.

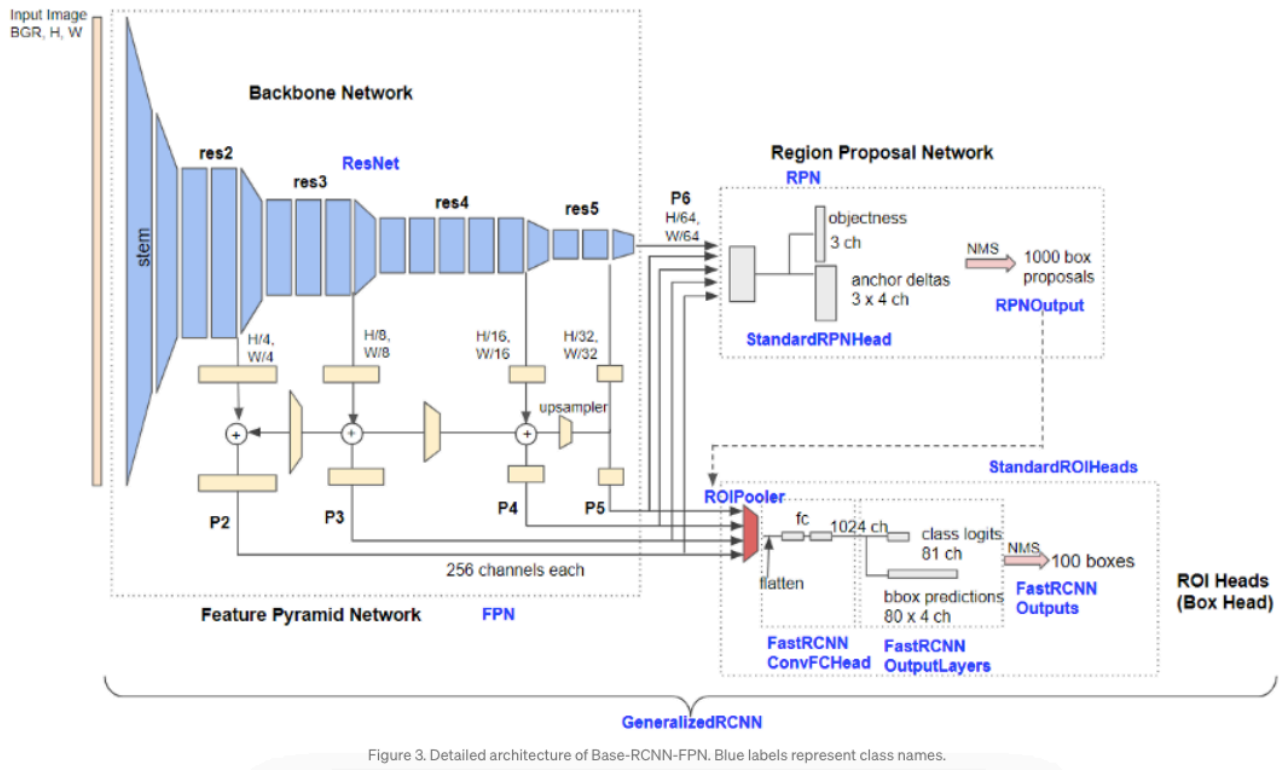


Figure 3. Detailed architecture of Base-RCNN-FPN. Blue labels represent class names.

Fig 2: Detailed architecture of Base-RCNN-FPN. Blue labels represent class names.

Structure of the detectron 2 repo:

The following is the directory tree of detectron 2 (under the 'detectron2' directory). Please just look at the 'modeling' directory. The Base-RCNN-FPN architecture is built by the several classes under the directory.

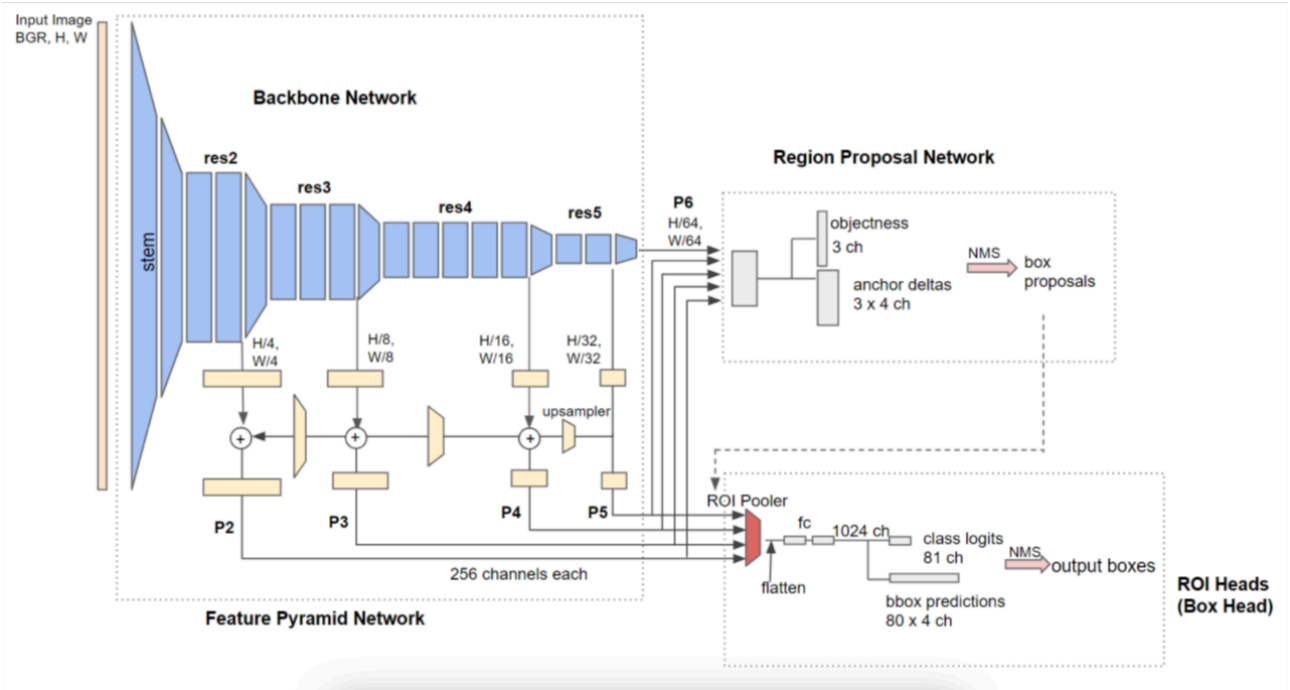
detectron2
- ├──**checkpoint** *<- checkpointer and model catalog handlers*
- ├──**config** *<- default configs and handlers*
- ├──**data** *<- dataset handlers and data loaders*
- ├──**engine** *<- predictor and trainer engines*
- ├──**evaluation** *<- evaluator for each dataset*
- ├──**export** *<- converter of detectron2 models to caffe2 (ONNX)*
- ├──**layers** *<- custom layers e.g. deformable conv.*

```
├──model_zoo  <- pre-trained model links and handler
├──modeling
|   ├──meta_arch <- meta architecture e.g. R-CNN, RetinaNet
|   ├──backbone  <- backbone network e.g. ResNet, FPN
|   ├──proposal_generator <- region proposal network
|   └──roi_heads <- head networks for pooled ROIs e.g. box, mask heads
├──solver       <- optimizer and scheduler builders
├──structures   <- structure classes e.g. Boxes, Instances, etc
└──utils        <- utility modules e.g. visualizer, logger, etc
```

Meta Architecture: GeneralizedRCNN (meta_arch/rcnn.py) which has:

- **Backbone Network:**
  FPN (backbone/fpn.py)
  └ ResNet (backbone/resnet.py)
- **Region Proposal Network**:
  RPN(proposal_generator/rpn.py)
  ├ StandardRPNHead (proposal_generator/rpn.py)
  └ RPNOutput (proposal_generator/rpn_outputs.py)
- **ROI Heads (Box Head)**:
  StandardROIHeads (roi_heads/roi_heads.py)
  ├ ROIPooler (poolers.py)
  ├ FastRCNNConvFCHead (roi_heads/box_heads.py)
  ├ FastRCNNOutputLayers (roi_heads/fast_rcnn.py)
  └ FastRCNNOutputs (roi_heads/fast_rcnn.py)

Each block has a main class and sub classes.
Now please see the *blue labels* on the Fig 2: You can see which class corresponds to which part of the pipeline. Here I add the architecture figure without class names. Below image is total architecture without class names.



The role of the backbone network is to extract feature maps from the input image. Here there are not any bounding boxes, anchors, or loss functions yet!

Input and Output of FPN:

Firstly we will clarify input and output of FPN. Fig 3 is the closer look at the FPN schematic.

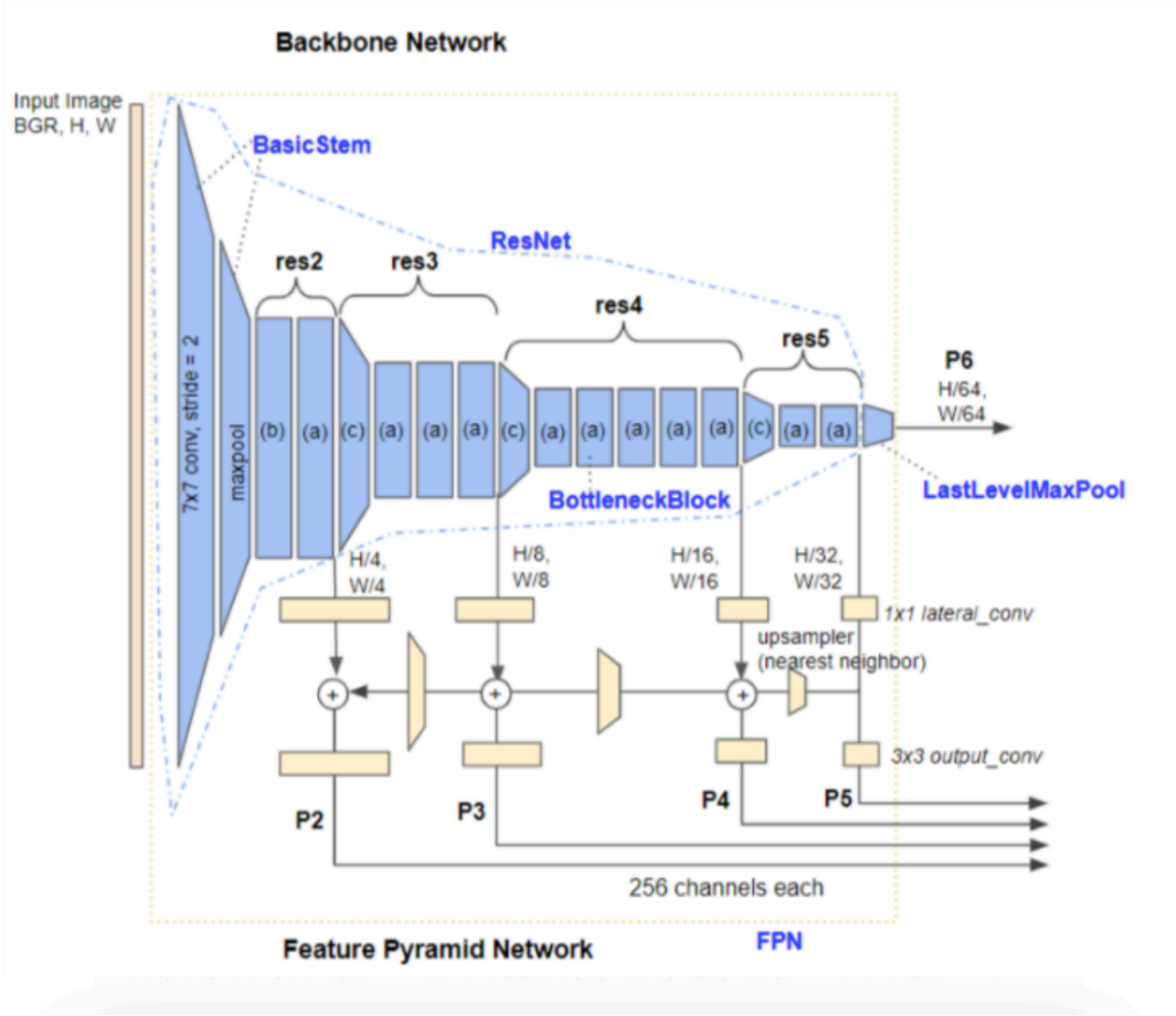


Fig 3. Detailed architecture of the backbone of Base-RCNN-FPN with ResNet50. Blue labels represent class names. (a), (b) and (c) inside the blocks stand for the bottleneck types detailed in Fig 5.

input (torch.Tensor): (B, 3, H, W) image:
B, H and W stand for batch size, image height and width respectively. Be careful that the order of input color channels is Blue, Green and Red (BGR). If you put an RGB image as input, detection accuracy might drop.
Output (dict of torch.Tensor): (B, C, H / S, W / S) feature maps:
C and S stand for channel size and stride. By default C=256 for all the scales and S = 4, 8, 16, 32 and 64 for P2, P3, P4, P5 and P6 outputs respectively.

For example, if we put a single image whose size is (H=800, W=1280) into the backbone, the input tensor size is `torch.Size([1, 3, 800, 1280])` and the output dict should be:

```
output["p2"].shape -> torch.Size([1, 256, 200, 320]) # stride = 4
output["p3"].shape -> torch.Size([1, 256, 100, 160]) # stride = 8
output["p4"].shape -> torch.Size([1, 256, 50, 80])   # stride = 16
output["p5"].shape -> torch.Size([1, 256, 25, 40])   # stride = 32
output["p6"].shape -> torch.Size([1, 256, 13, 20])   # stride = 64
```

Fig 4 shows what the actual output feature maps look like. One pixel of 'P6' feature corresponds to broader area of input image than 'P2'- in other words 'P6' has a larger **receptive field** than 'P2'. FPN can extract multi-scale feature maps with different receptive fields.

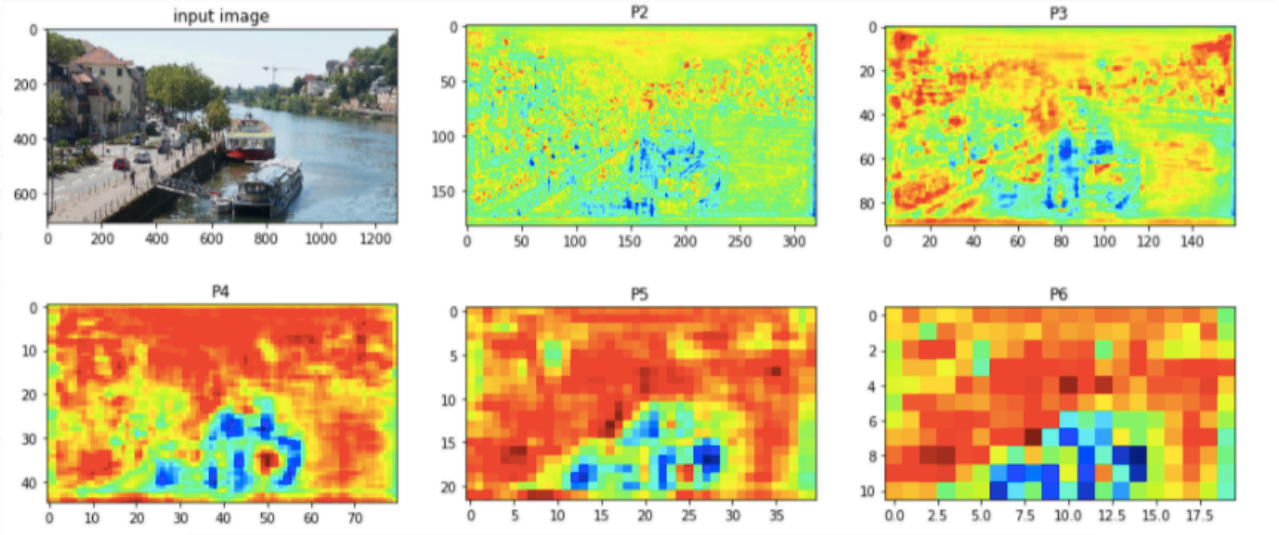


Fig 4: Example of input and output of FPN. The feature at the 0th channel is visualized from each output.

Code Structure[4] for Backbone modeling

The related files are under `detectron2/modeling/backbone` directory:

```
├─modeling
│   ├─backbone
│   │    ├─backbone.py. <- includes abstract base class Backbone
│   │    ├─build.py     <-call builder function specified in config
│   │    ├─fpn.py       <- includes FPN class and sub-classes
│   │    ├─resnet.py    <- includes ResNet class and sub-classes
```

The following is the class hierarchy.

```
FPN (backbone/fpn.py)
├ ResNet (backbone/resnet.py)
|   ├ BasicStem (backbone/resnet.py)
|   └ BottleneckBlock (backbone/resnet.py)
└ LastLevelMaxPool (backbone/fpn.py
```

ResNet

ResNet[5] consists of a stem block and 'stages' that contain multiple bottleneck blocks. As for ResNet50, the block structure is :

```
BasicStem
(res2 stage, 1/4 scale)
BottleneckBlock (b)(stride=1, with shortcut conv)
BottleneckBlock (a)(stride=1, w/0  shortcut conv) × 2
(res3 stage, 1/8 scale)
BottleneckBlock (c)(stride=2, with shortcut conv)
BottleneckBlock (a)(stride=1, w/o  shortcut conv) × 3
(res4 stage, 1/16 scale)
BottleneckBlock (c)(stride=2, with shortcut conv)
BottleneckBlock (a)(stride=1, w/o shortcut conv) × 5
(res5 stage, 1/32 scale)
BottleneckBlock (c)(stride=2, with shortcut conv)
BottleneckBlock (a)(stride=1, w/o  shortcut conv) × 2
```

ResNet101 and ResNet152 have larger number of bottleneck blocks (a), defined at: [code link].

(1) BasicStem (stem block) [code link]

The 'stem' block of ResNet is quite simple. It down-samples the input image twice by 7×7 convolution with stride=2 and max pooling with stride=2.
The output of the stem block is a feature map tensor whose size is (B, 64, H / 4, W / 4).

```
- conv1 (kernel size = 7, stride = 2)
- batchnorm layer
- ReLU
- maxpool layer (kernel size = 3, stride = 2)
```

(2) BottleneckBlock [code link]

The bottleneck block is originally proposed in the ResNet paper[5]. The block has three convolution layers whose kernel sizes are 1×1, 3×3, 1×1 respectively. The input and output channel numbers of 3×3 convolution layer are smaller than the input and output of the block, for efficient computation.

There are three types of bottleneck blocks as shown in Fig 5 :
(a): stride=1, w/o shortcut conv
(b): stride=1, with shortcut conv
(c) : stride=2, with shortcut conv

shortcut convolution (used in (b), (c))

ResNet has identity shortcut that adds the input and the output features. For the first block of a stage (*res2-res5*), a shortcut convolution layer is used to match the number of channels of input and output.
downsampling convolution with stride=2 (used in (c))

At the first block of the *res3*, *res4* and *res5* stages, the feature map is downsampled by a convolution layer with stride=2. A shortcut convolution with stride=2 is also used, because the input channel number is not the same as the output.
Note that a 'convolution layer' mentioned above contains convolution `torch.nn.Conv2d` and normalization (e. g. FrozenBatchNorm[6]). [code link]
ReLU activation is used after convolutions and feature addition (see Fig 5).

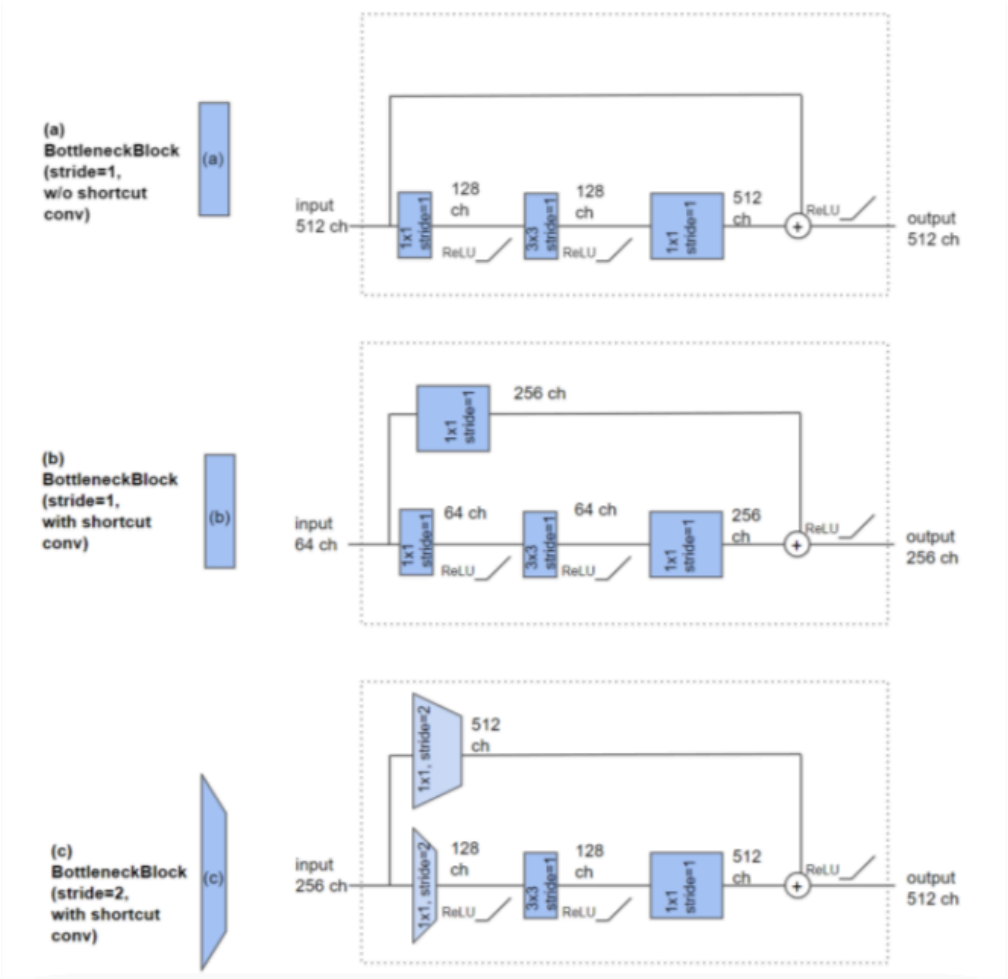


Fig 5. Three types of bottleneck blocks.

FPN:

FPN contains ResNet, lateral and output convolution layers, up-samplers and a last-level maxpool layer. [code link]

lateral convolution layers [code link]
This layer is called 'lateral' convolution because FPN is originally depicted like a pyramid where the stem layer is placed at the bottom (it is rotated in this article). The lateral convolution layers take features from the *res2-res5* stages with different channel numbers and returns 256-ch feature maps.

output convolution layers [code link]
An output convolution layer contains 3×3 convolution that does not change number of channels.

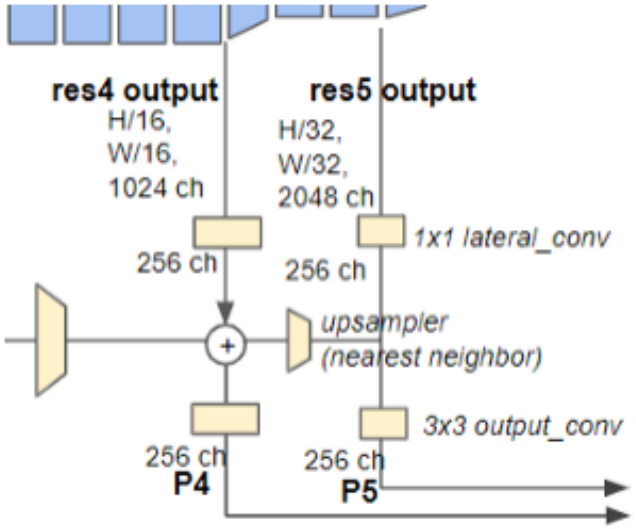


Fig 6. Zooming into a part of FPN schematic that deals with res4 and res5.

forward process [code link]
H/32, The forward process of FPN starts from the *res5* output (see Fig 6). After going through the lateral convolution, the 256-channel feature map is fed to the output convolution, to be registered to the *results* list as P5 (1/32 scale). The 256-channel feature map is also fed to the up-sampler (`F.interpolate` with nearest neighbor) and added to the res4 output (via lateral convolution). The resulting feature map goes through the output convolution and the result tensor P4 is inserted to the *results* list (1/16 scale).

The routine above (from up-sampling to insertion to the *results*) is carried out three times, and finally the *result* list contains *four* tensors — namely P2 (1/4 scale), P3 (1/8), P4 (1/16) and P5 (1/32).

LastLevelMaxPool [code link]
To make the P6 output, a max pooling layer with kernel size = 1 and stride = 2 is added to the final block of the ResNet. This layer just down-samples P5 features (1/32 scale) to 1/64-scale features to be added to the *result* list.

Now we have obtained multi-scale feature maps. In the next part we will look at the data loader and ground-truth data preparation.

I have shown the details of feature pyramid network (FPN).
Before proceeding to the Region Proposal Network (RPN), we should understand the data structure of ground truth. In this part I'm going to share how to load ground truth from a dataset and how the loaded data are processed before being fed to the network.

Where in the network are ground truth data used ?

To train a detection model, you need to prepare **images and annotations.**
As for the Base-RCNN-FPN (Faster R-CNN), the ground truth data are used in the Region Proposal Network (RPN) and the Box Head (see Fig 7).

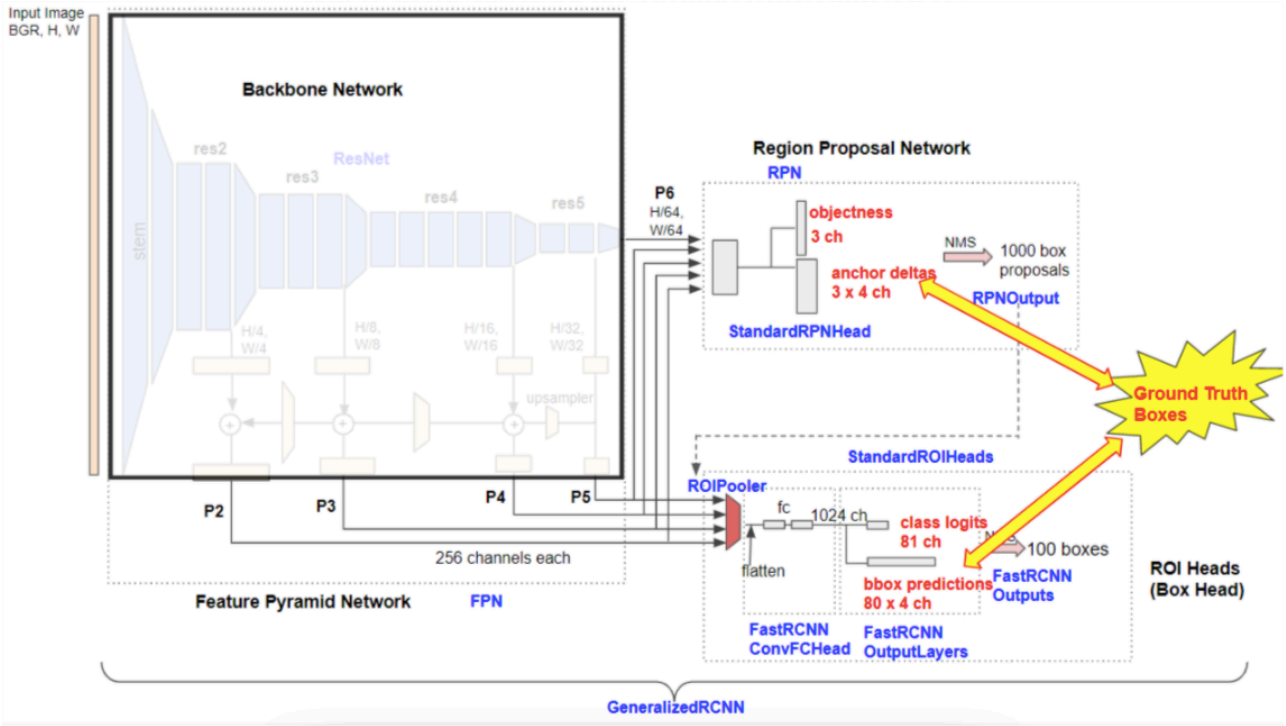


Fig 7. Ground truth box annotations are used in the Region Proposal Network and Box Head to calculate losses.

Annotation data for object detection consist of:
Box label : location and size of the object (e.g. [x, y, w, h])
Category label: object class id (e.g. 12: "parking meter")
Note that the RPN does not learn to classify object categories, so category labels are used only at the ROI Heads.

The ground truth data are loaded from the annotation file of the specified dataset. Let's look at the process of data loading.

Data loader

The data loader of Detectron 2 is multi-level nested. It is built by the builder before starting training[3].

- *dataset_dicts (list)* is a list of annotation data registered from the dataset.
- *DatasetFromList (data.Dataset)* takes a *dataset_dicts* and wrap it as a torch dataset.
- *MapDataset (data.Dataset)* calls *DatasetMapper* class to map each element of DatasetFromList. It loads images, transforms images and annotations, and converts annotations to an 'Instances' object.

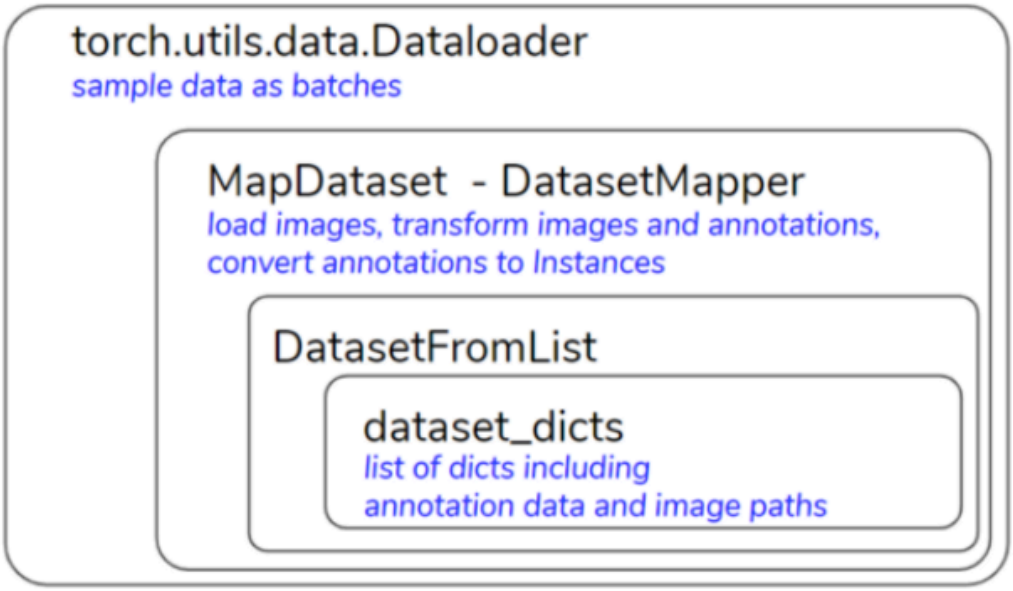


Fig 8.Data loader of Detectron 2.

Loading annotation data

Let's assume we have a dataset called '*mydataset*' with the following image and annotations[4].

To load data from a dataset, it must be registered to DatasetCatalog. For instance, to register *mydataset*,

```
from detectron2.data import DatasetCatalog
from mydataset import load_mydataset_json
def register_mydataset_instances(name, json_file):
    DatasetCatalog.register(name, lambda:
load_mydataset_json(json_file, name))
```

and call *register_mydataset_instances* function specifying your json file path.

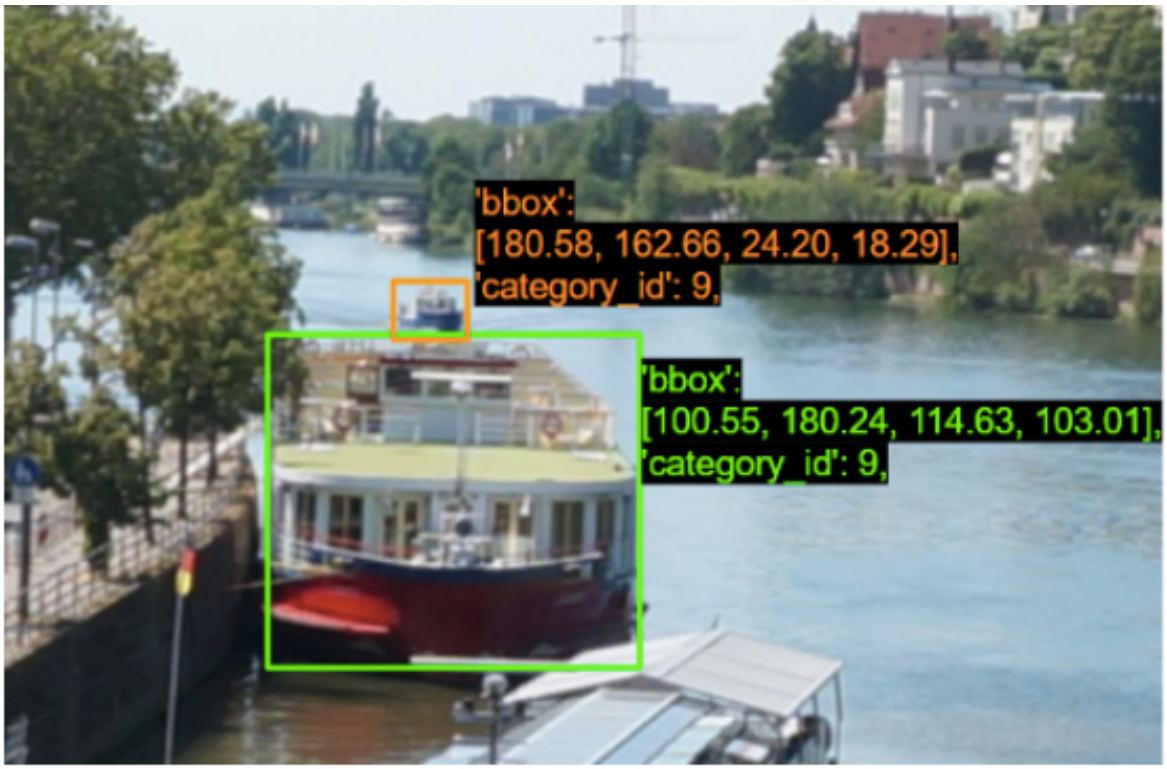


Fig 9. Example of an image and annotations

The *load_mydataset_json* function must include a json loader so that the following list of dict records is returned:

```
[
{
'file_name': 'imagedata_1.jpg',   # image file name
'height': 640,                     # image height
'width': 640,                      # image width
'image_id': 12,                    # image id
'annotations': [                   # list of annotations
{'iscrowd': 0,                           # crowd flag
'bbox': [180.58, 162.66, 24.20, 18.29],  # bounding box label
'category_id': 9,                        # category label
'bbox_mode': <BoxMode.XYWH_ABS: 1>}      # box coordinate mode
,...
]
},
```

For COCO dataset (Detectron 2's default), *load_coco_json* function plays the role.

Mapping data

During training, registered annotation records are picked one by one. We need actual image data (not path) and corresponding annotations. The dataset mapper (*DatasetMapper*) deals with the records to add an 'image' and 'Instances' to the *dataset_dict*. 'Instances' are the ground truth structure object of Detectron 2.

- Load and transform images
  An image specified by 'file name' is loaded by read_image function. Loaded image is transformed by pre-defined transformers (such as left-right flip) and finally the image tensor whose shape is (channel, height, width) is registered.
- Transform annotations
  The 'annotations' of dataset_dict are transformed by the transformations performed on the images. For example, if the image has been flipped, the box coordinates are changed to the flipped location.
- Convert annotations to Instances
  The annotations are converted to Instances by this function called in the dataset mapper. 'bbox' annotations are registered to *Boxes* structure object which can store a list of bounding boxes. 'category_id' annotations are simply convereted to a torch tensor.

After mapping, the *dataset_dict* looks like:

```
{'file_name': 'imagedata_1.jpg',
'height': 640, 'width': 640, 'image_id': 0,
'image': tensor([[[255., 255., 255.,  ...,  29.,  34.,
36.],...[169., 163., 162.,  ...,  44.,  44.,  45.]]]),
'instances': {
'gt_boxes': Boxes(tensor([[100.55, 180.24, 114.63, 103.01],
[180.58, 162.66, 204.78, 180.95]])),
'gt_classes': tensor([9, 9]),
}
```

Now we have images and ground-truth annotations which Detectron 2 models can learn.

In the next part we will see how region proposal network learns object locations.

This time, we are going deep into the most complicated but important part — the Region Proposal Network.
As we have already seen in the output feature maps from the feature pyramid network are:

```
output["p2"].shape -> torch.Size([1, 256, 200, 320]) # stride = 4
output["p3"].shape -> torch.Size([1, 256, 100, 160]) # stride = 8
output["p4"].shape -> torch.Size([1, 256, 50, 80]) # stride = 16
output["p5"].shape -> torch.Size([1, 256, 25, 40]) # stride = 32
output["p6"].shape -> torch.Size([1, 256, 13, 20]) # stride = 64
```

which are also the input to the RPN. Each tensor size stands for (batch, channels, height, width). We use the feature dimensions above throughout this blog part.

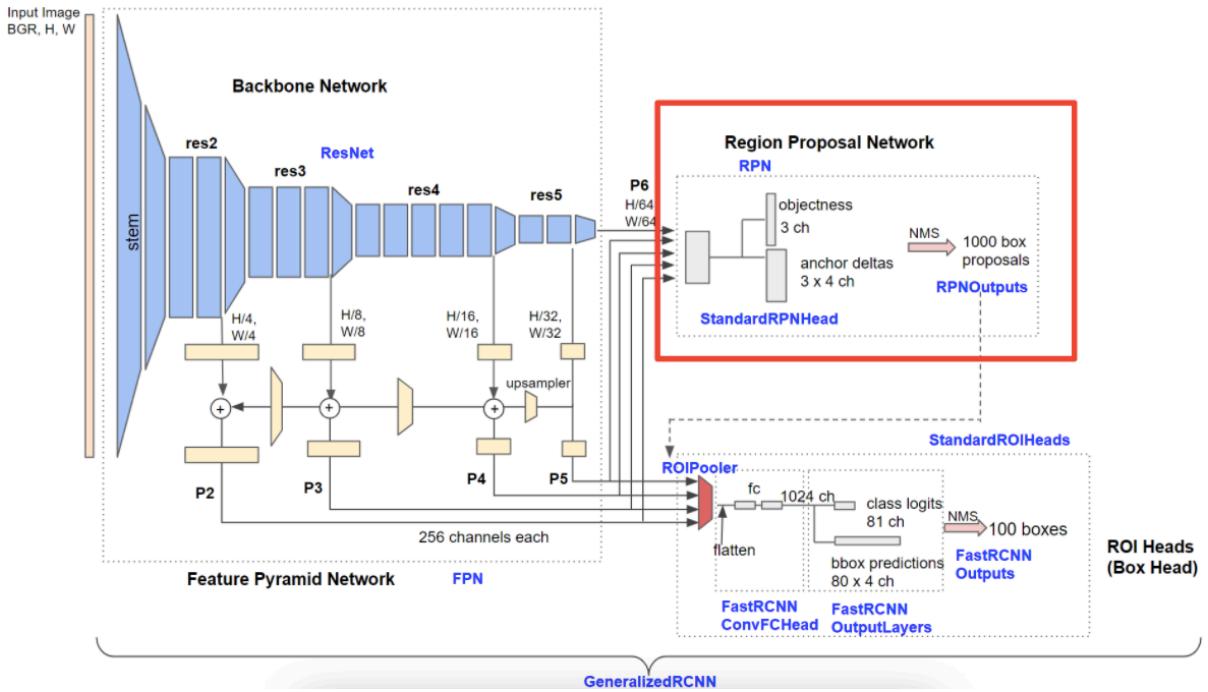


Fig 10. Detailed architecture of Base-RCNN-FPN. Blue labels represent class names.

We also have ground truth boxes loaded from the dataset:

```
'gt_boxes': Boxes(tensor([
[100.58, 180.66, 214.78, 283.95],
[180.58, 162.66, 204.78, 180.95]
])),
'gt_classes': tensor([9, 9]) # not used in RPN!
```

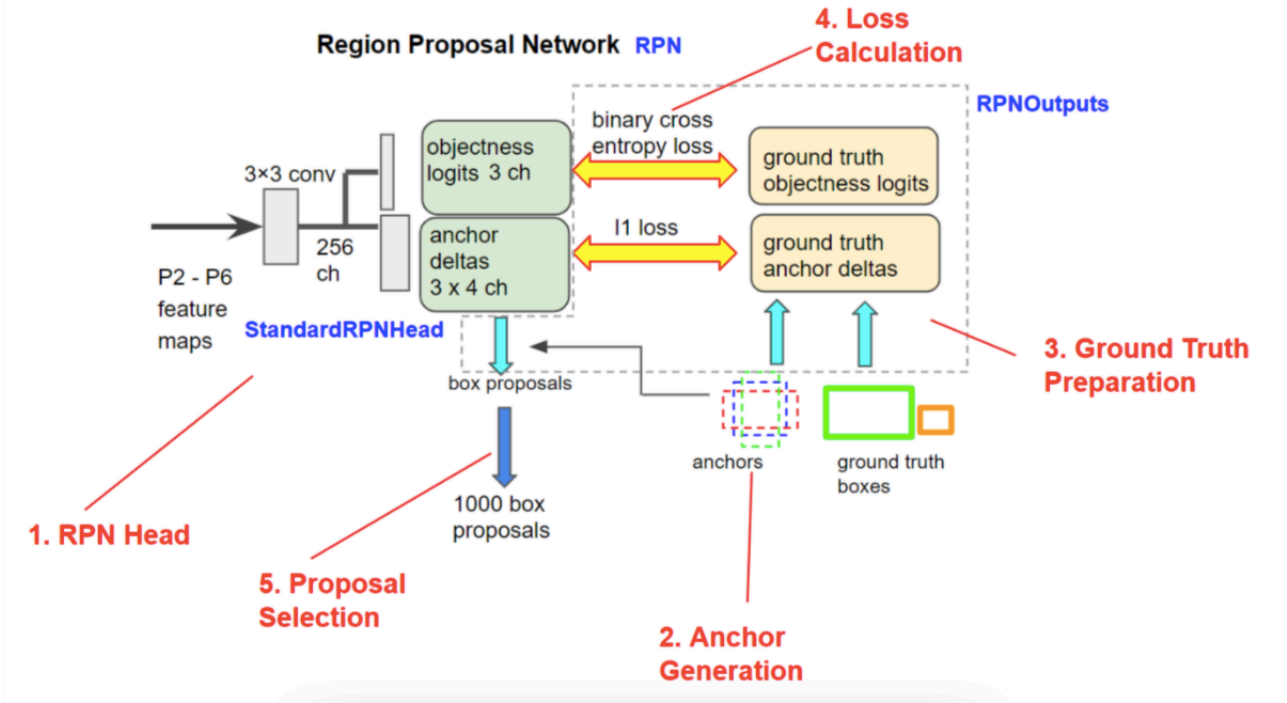


Fig 11. Schematic of Region Proposal Network. Blue and red labels represent class names and chapter titles respectively.

How can object detectors connect the feature maps and ground-truth box locations and sizes? Let's see how RPN — the core component of RCNN detector — works.

Fig 11 shows the detailed schematic of RPN. RPN consists of a neural network (RPN Head) and non-neural-network functionalities. All the computation in RPN[3] is performed on GPU in Detectron 2.

Firstly, let's see the RPN Head that processes the feature maps fed from the FPN.

1. RPN Head

The neural network part of RPN is simple. It is called RPN Head and consists of three convolution layers defined in the StandardRPNHead class.

1. *conv (3×3, 256 -> 256 ch)*
2. *objectness logits conv (1×1, 256 -> 3 ch)*
3. *anchor deltas conv (1×1, 256 -> 3×4 ch)*

The feature maps of five levels (P2 to P6) are fed to the network one by one. The output feature maps at one level are:

1. *pred_objectness_logits (B, 3 ch, Hi, Wi): probability map of object existence*
2. *pred_anchor_deltas (B, 3×4 ch, Hi, Wi): relative box shape to anchors*

where B stands for batch size and Hi and Wi correspond to the feature map sizes of P2 to P6.

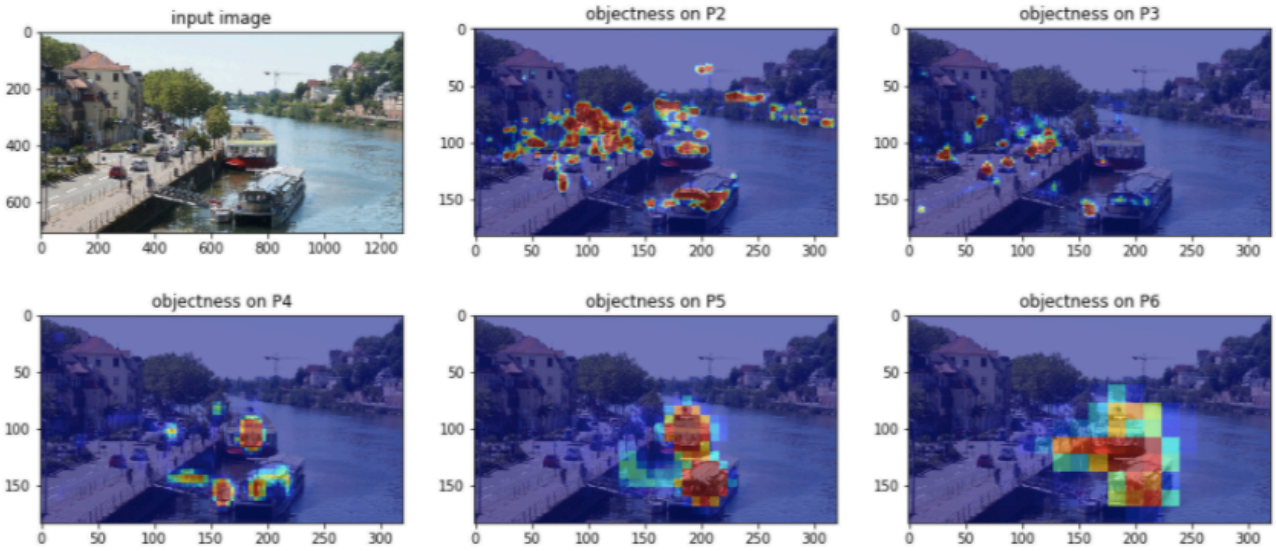


Fig 12. Visualization of objectness maps. Sigmoid function has been applied to the objectness_logits map. The objectness maps for 1:1 anchor are resized to the P2 feature map size and overlaid on the original image.

What do they look like actually? In Fig 12, the objectness logits map at each level is overlaid on the input image. You can find that small objects are detected at P2 and P3 and the larger ones at P4 to P6. This is exactly what feature pyramid network aims for. The multi-scale network can detect tiny objects which a single-scale detector cannot find.

Next, let's proceed to anchor generation, which is essential to associate the ground truth boxes with the two output feature maps above.

2. Anchor Generation

To connect the objectness map and anchor deltas map to the ground truth boxes, the reference boxes called 'anchors' are necessary.

2–1. Generate Cell Anchors

In Base-FPN-RCNN of Detectron 2, the anchors are defined as follows:

```
MODEL.ANCHOR_GENERATOR.SIZES = [[32], [64], [128], [256], [512]]
MODEL.ANCHOR_GENERATOR.ASPECT_RATIOS = [[0.5, 1.0, 2.0]]
```

What does it mean?

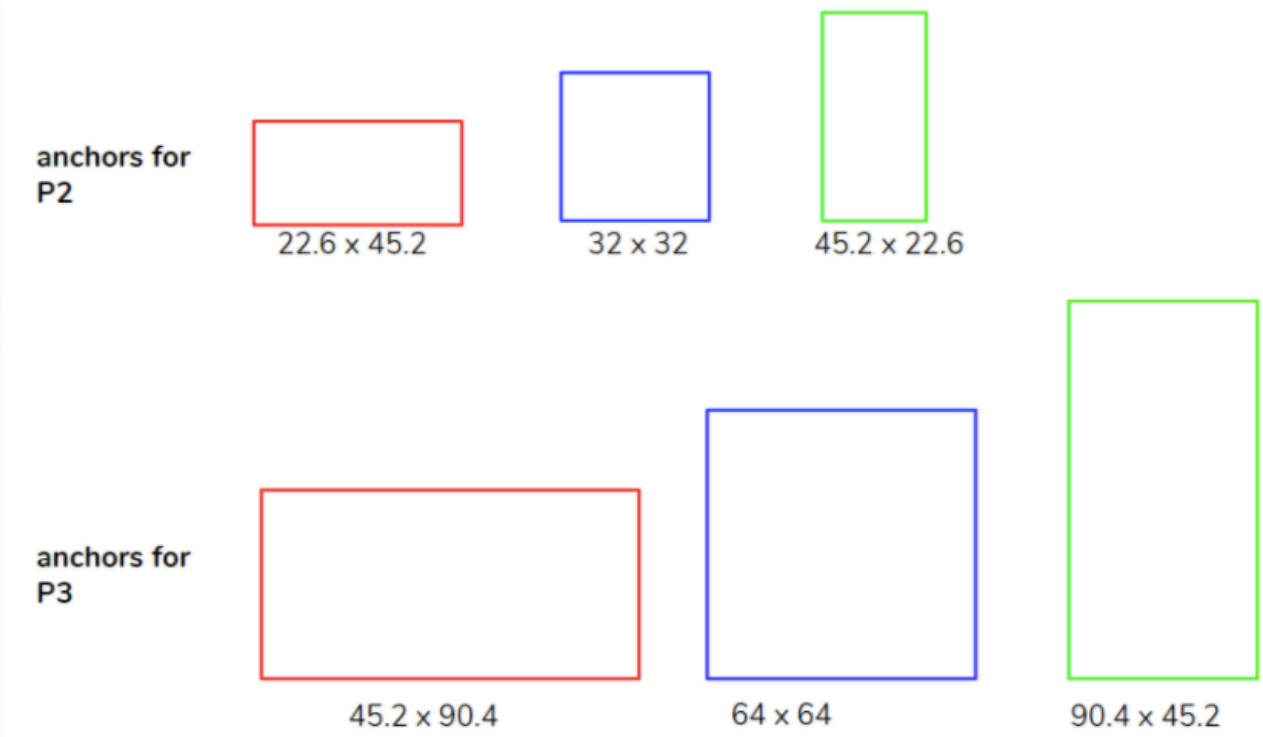


Fig 13. Cell anchors for P2 and P3 feature maps. (from left: 1:2, 1:1 and 2:1 aspect ratios)

The five elements of ANCHOR_GENERATOR.SIZES list correspond to five levels of feature maps (P2 to P6). For example P2 (stride=4) has one anchor whose size is 32.

Aspect ratios define the shapes of anchors. For the example above, there are three shapes: 0.5, 1.0 and 2.0. Let's see the actual anchors (Fig. 13). The three anchors at P2 feature map have aspect ratios of 1:2, 1:1 and 2:1 and the same areas as 32 ×32. At P3 level anchors are twice as large as the P2 anchors.

These anchors are called 'cell anchors' in Detectron 2. (The code for anchor generation is here.) As a result we have obtained 3×5=15 cell anchors for the five feature map levels.

```
# cell anchors for P2, P3, P4, P5 and P6. (x1, y1, x2, y2)
tensor([[-22.6274, -11.3137,  22.6274,  11.3137],
        [-16.0000, -16.0000,  16.0000,  16.0000],
        [-11.3137, -22.6274,  11.3137,  22.6274]])
tensor([[-45.2548, -22.6274,  45.2548,  22.6274],
        [-32.0000, -32.0000,  32.0000,  32.0000],
        [-22.6274, -45.2548,  22.6274,  45.2548]])
tensor([[-90.5097, -45.2548,  90.5097,  45.2548],
        [-64.0000, -64.0000,  64.0000,  64.0000],
        [-45.2548, -90.5097,  45.2548,  90.5097]])
tensor([[-181.0193,  -90.5097,  181.0193,   90.5097],
        [-128.0000, -128.0000,  128.0000,  128.0000],
        [ -90.5097, -181.0193,   90.5097,  181.0193]])
tensor([[-362.0387, -181.0193,  362.0387,  181.0193],
        [-256.0000, -256.0000,  256.0000,  256.0000],
        [-181.0193, -362.0387,  181.0193,  362.0387]])
```

2–2. Place Anchors on the Grid Points

Next we place the cell anchors on the grid cells whose sizes are the same as predicted feature maps.

For example our predicted feature map 'P6' has the size of (13, 20) and stride of 64. In Fig. 6 the P6 grid is shown with three anchors placed at (5, 5). In the input image resolution, the coordinate of (5, 5) corresponds to (320, 320) and the square anchor's size is (512, 512). The anchors are placed at every grid point, so 13×20×3 = 780 anchors are generated for P6.

The same process is carried out for the other grid points (see the example of P5 grid in Fig 14) and totally 255,780 anchors are generated.

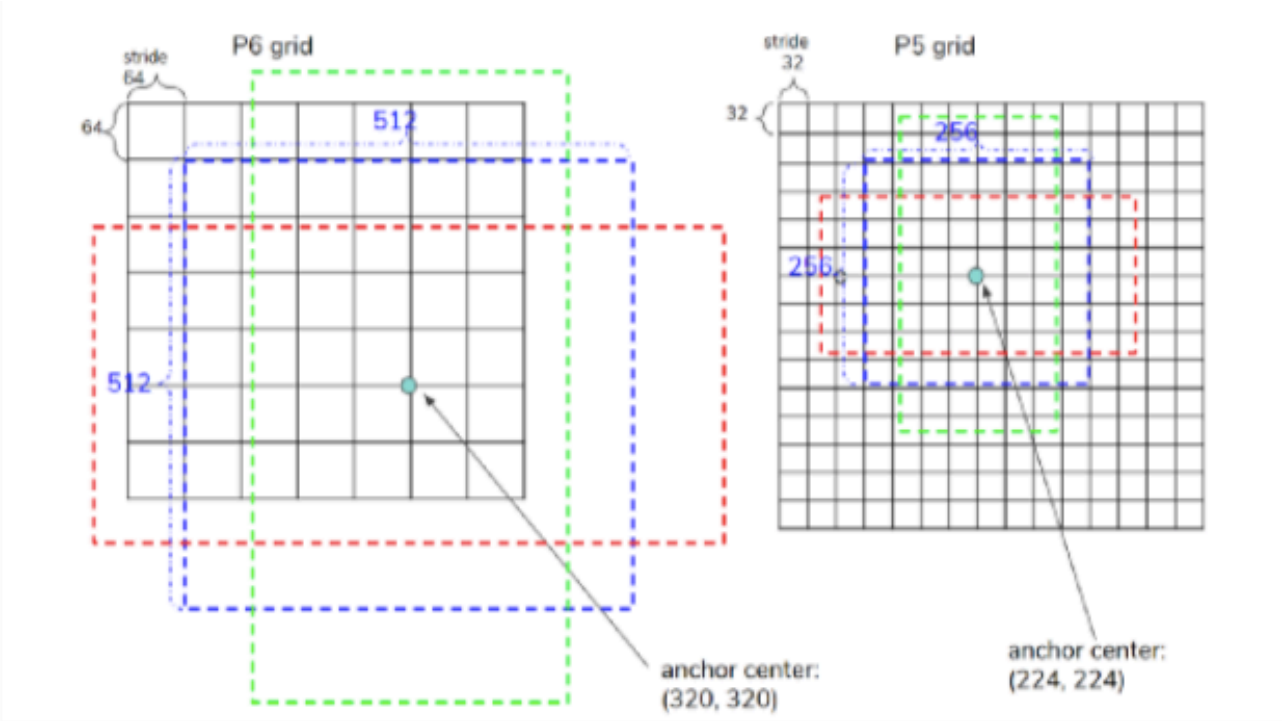


Fig 14 Placing anchors on grid points. The top-left corner of each grid corresponds to(0, 0)

3. Ground Truth Preparation

In this chapter we associate ground truth boxes with the generated anchors.

3–1. Calculate Intersection-over-Unit (IoU) Matrix

Let's assume we have two ground truth (GT) boxes loaded from a dataset.

```
'gt_boxes':
Boxes(tensor([
[100.58, 180.66, 214.78, 283.95],
[180.58, 162.66, 204.78, 180.95]
])),
```

We now try to find the anchors similar to the two GT boxes out of the 255780 anchors. How can you tell whether a box is similar to another? The answer is Intersection over Union (IoU) calculation. In Detectron2, pairwise_iou function can calculate IoU for every pair from two lists of boxes. In our case, result of pairwise_iou is a matrix whose size is (2(GT), 255780(anchors)).

```
# Example of IoU matrix, the result of pairwise_iou
tensor([[0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000],#GT 1
        [0.0000, 0.0000, 0.0000, ..., 0.0087, 0.0213, 0.0081],#GT 2
```

3–2. Examine the IoU matrix by Matcher

The IoU matrix is examined by Matcher and all the anchors are labeled as foreground, background, or ignored. As shown in Fig 15, if IoU is larger than

the pre-defined threshold (typically 0.7), the anchor is assigned to one of the GT boxes and labeled as foreground ('1'). If IoU is smaller than another threshold (typically 0.3), the anchor is labeled as a background ('0'), otherwise ignored ('-1').

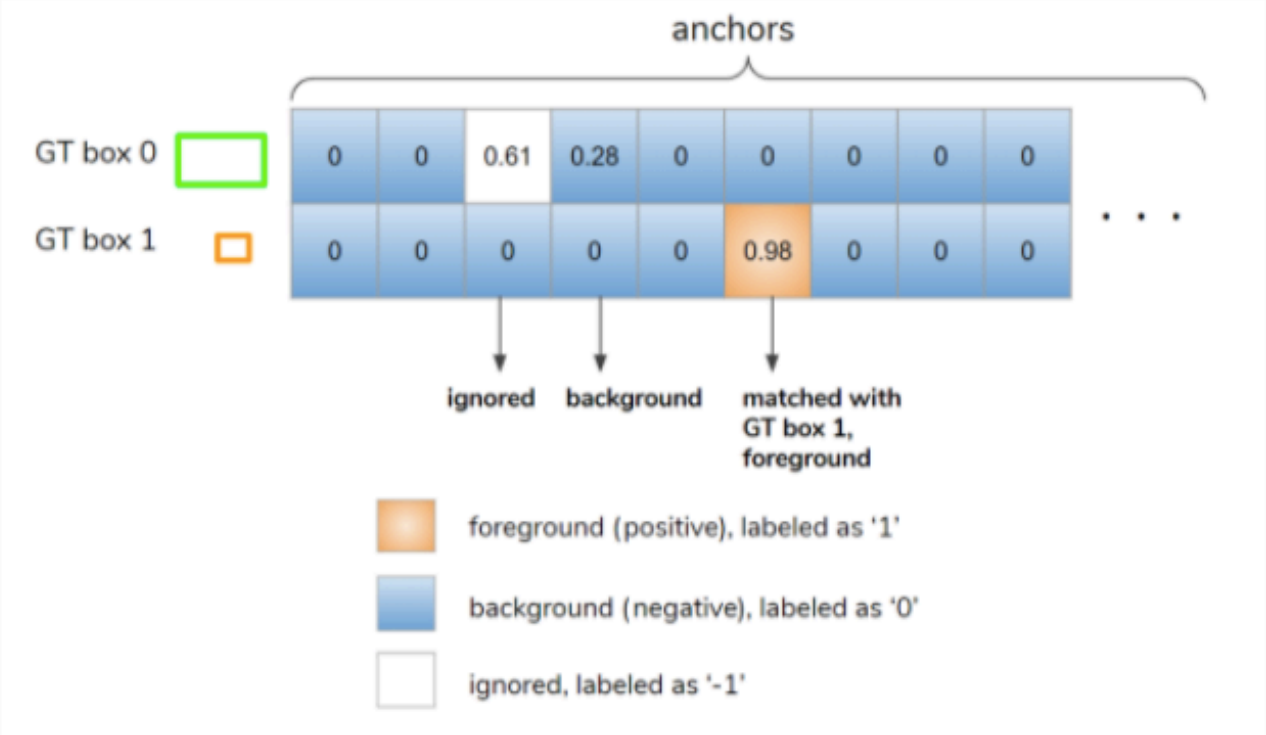


Fig 15. Matcher determines assignment of anchors to ground-truth boxes. The table shows the IoU matrix whose shape is (number of GT boxes, number of anchors).

In Fig 16 we show the matching result overlaid on the input image. As you can see, most of the grid points are labeled as background (0) and a few of them as foreground (1) and ignored (-1).

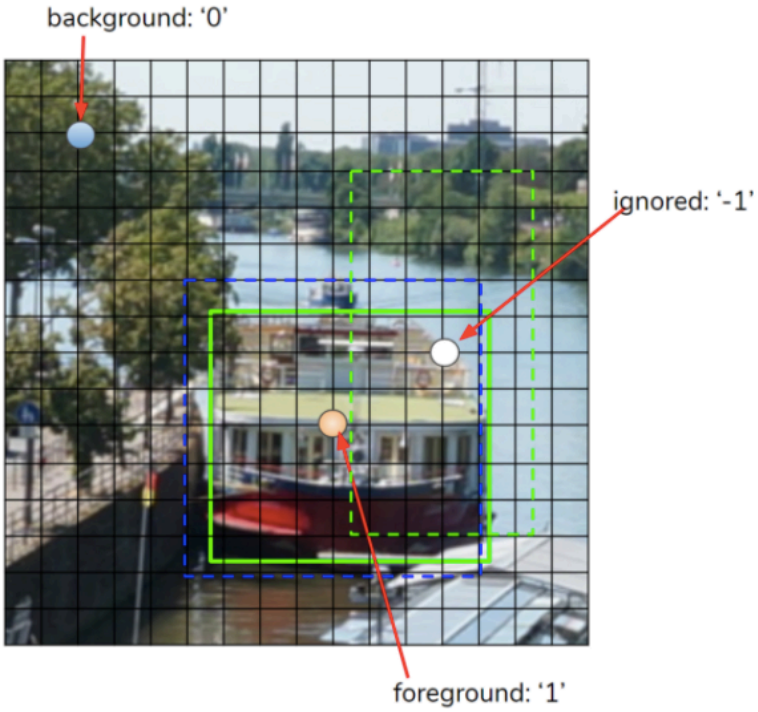


Fig 16. Matching result overlaid on the input image.

3–3. Calculate anchor deltas

The anchor boxes determined as foreground have similar shapes to the GT boxes. However the network should learn to propose the exact locations and shapes of GT boxes. To achieve it four regression parameters should be learned : Δx, Δy, Δw, and Δh. These 'deltas' are calculated as shown in Fig. 9 using the Box2BoxTransform.get_deltas function. The formulation is written in the Faster-RNN paper[4].

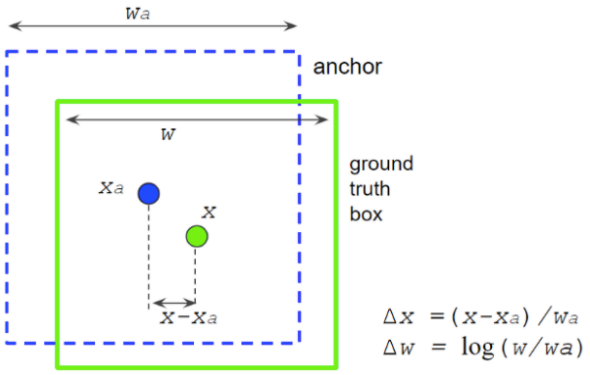


Fig 16. Anchor deltas calculation.

As a result, we obtain a tensor called *gt_anchor_deltas* whose shape is (255,780, 4) in our case.

```
# calculated deltas (dx, dy, dw, dh)
tensor([[ 9.9280, 24.6847,  0.8399,  2.8774],
        [14.0403, 17.4548,  1.1865,  2.5308],
        [19.8559, 12.3424,  1.5330,  2.1842],
        …,
```

3–4. Re-sample the boxes for loss calculation

Now we have objectness_logits and anchor_deltas on every grid point of feature maps, which we can compare predicted feature maps with.

In Fig.17 (left) is the breakdown of number of anchors per image and example. As you can see, the majority of anchors are background. For example, typically there are less than 100 foreground anchors, less than 1000 ignored anchors in 255,780 anchors and the rest are background. If we go on training, it's hard to learn the foreground ones due to the label imbalance.

The labels are re-sampled by using the subsample_labels function to solve the imbalance issue.

Let N be the target number of foreground + background boxes and F be the target number of foreground boxes. N and F / N are defined by the following config parameters.

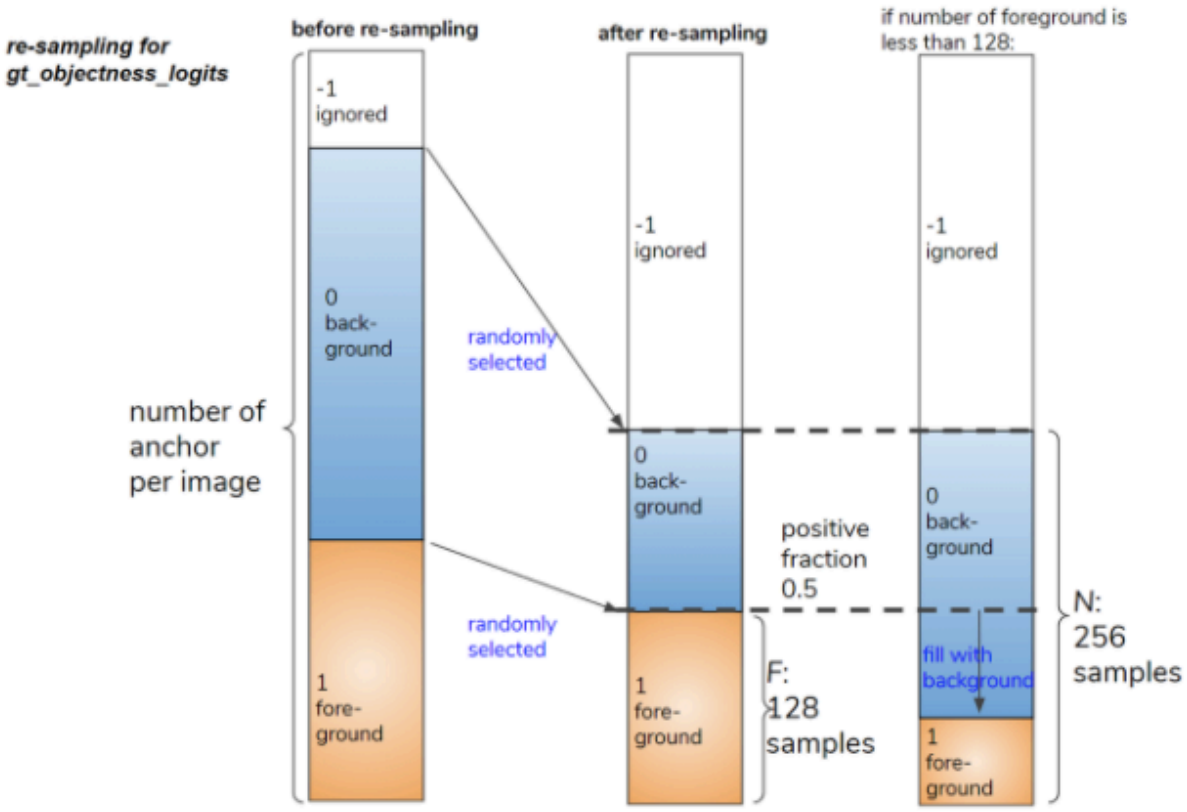


Fig 17. Re-sampling the foreground and background boxes.

`N: MODEL.RPN.BATCH_SIZE_PER_IMAGE (typically 256)`
`F/N: MODEL.RPN.POSITIVE_FRACTION (typically 0.5)`

Fig 17 (center) shows the breakdown of re-sampled boxes. Background and foreground boxes are randomly selected so that N and F/N become the values defined by the parameters above. In case foreground number is less than F as shown in Fig 17 (right), background boxes are sampled to fill the N samples.

4. Loss Calculation

Two loss functions are applied to the prediction and ground truth maps at the rpn_losses function.

localization loss (loss_rpn_loc)

- l1 loss[5].
- Calculated **only at the grid points where ground-truth objectness=1 (foreground)**, which means all the background grid points are ignored to compute the loss.

objectness loss (loss_rpn_cls)

- Binary cross entropy loss.
- Calculated **only at the grid points where ground-truth objectness=1 (foreground) or 0 (background)**.

The actual loss results are as follows:

```
{
'loss_rpn_cls': tensor(0.6913, device='cuda:0',
grad_fn=<MulBackward0>),
'loss_rpn_loc': tensor(0.1644, device='cuda:0',
grad_fn=<MulBackward0>)
}
```

5. Proposal Selection

Lastly we choose 1000 'region proposal' boxes from the predicted boxes following the four steps below.

- Apply predicted *anchor_deltas* to the corresponding anchors, which is the reverse process of 3–3.
- The predicted boxes are sorted by the predicted objectness scores at each feature level.
- As shown in Fig. 18, the top-K scored boxes (defined by the config parameters) are chosen from each feature level of an image[6]. For example, 2,000 boxes are chosen from 192,000 boxes at P2. For P6 where less than 2,000 boxes exist, all the boxes are selected.
- Non-maximum suppression (batched_nms) is applied at each level independently. 1,000 top-scored boxes survive as a result.

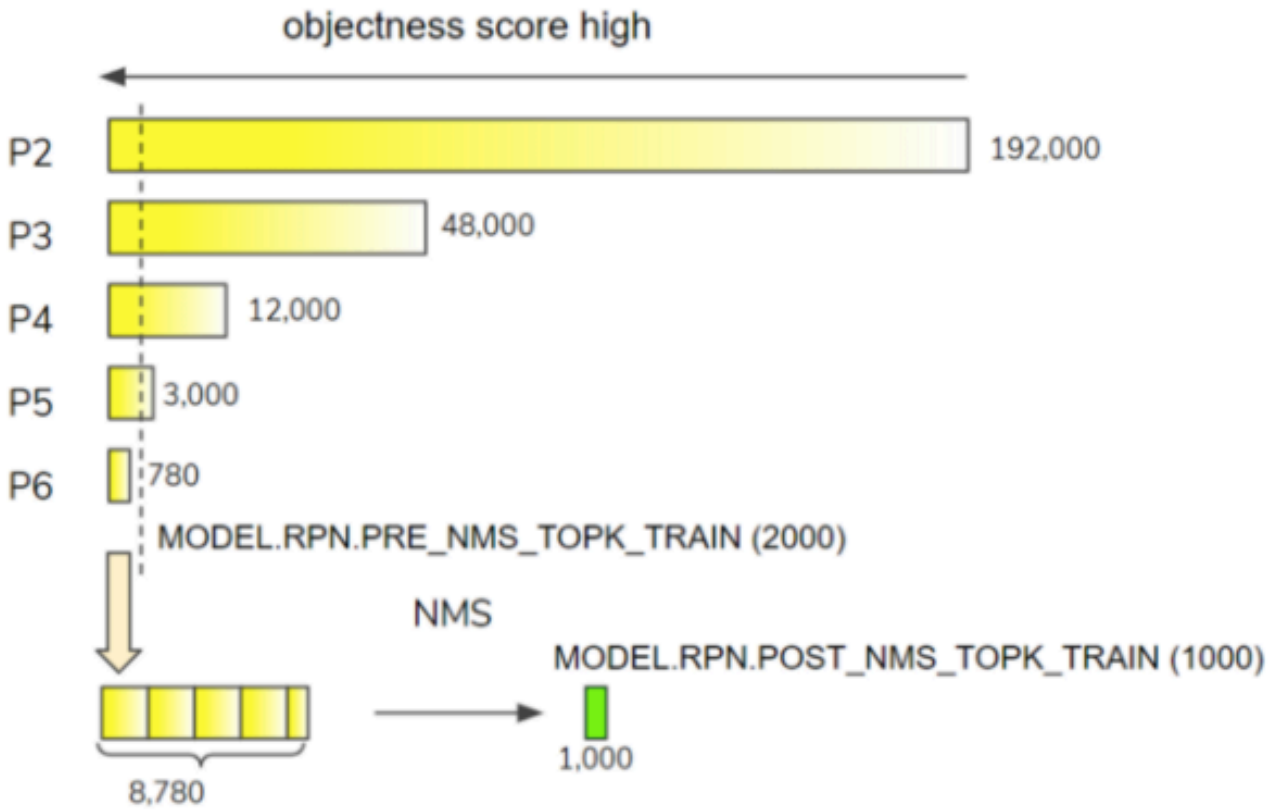


Fig 18. Choosing top-K proposal boxes from each feature level. The numbers of boxes are the examples when input image size is (H=800, W=1280).

Finally, we obtain proposal boxes as 'Instances' with:
'proposal_boxes': 1,000 boxes
'objectness_logits': 1,000 scores
which are used in the next stage.

In the next part we proceed to the Box head, the second stage of R-CNN.

Till now we have seen the overview of the Base-RCNN-FPN, feature pyramid network, ground truth preparation and region proposal network, respectively. This time, we are going deep into the final part of the pipeline— the ROI (Box) Head[3] (see Fig 19).

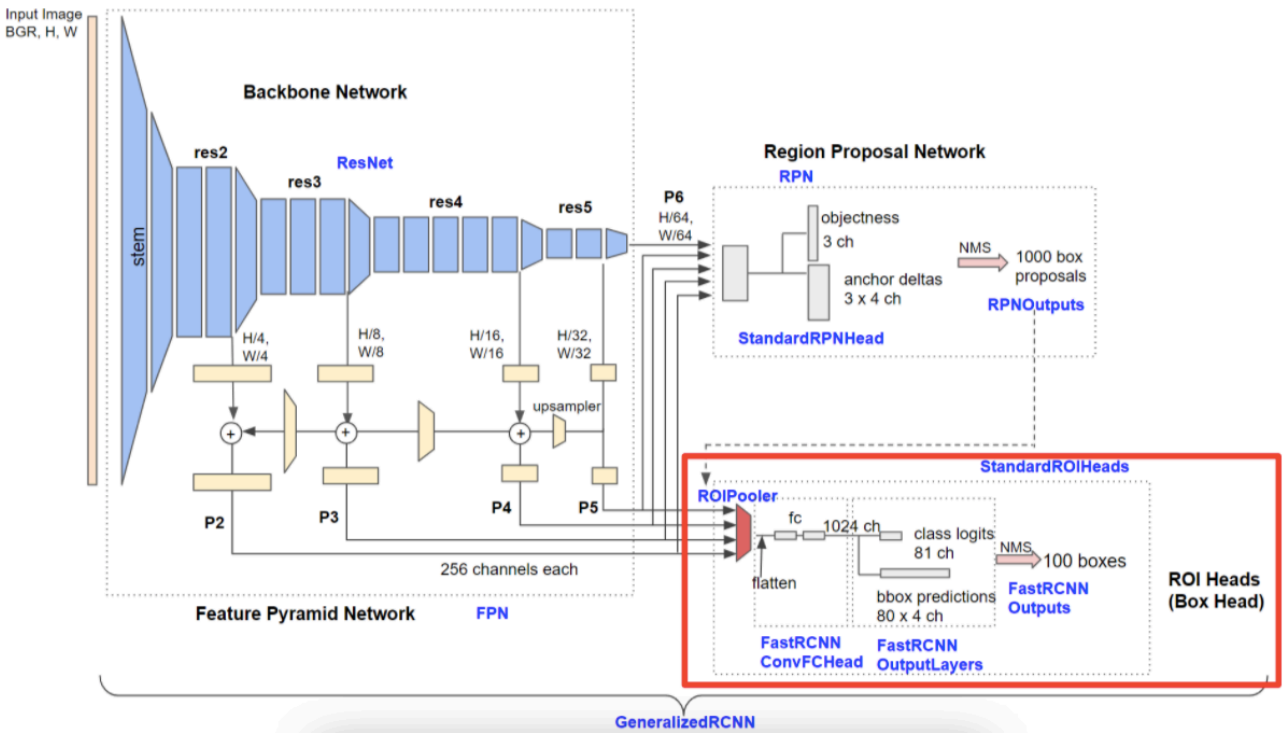


Fig 19. Detailed architecture of Base-RCNN-FPN. Blue labels represent class names.

At the ROI (Box) Head, we take a) feature maps from FPN, b) proposal boxes, and c) ground truth boxes as input.

a) **feature maps from FPN**
As we have seen in part 2, the output feature maps from FPN are:

```
output["p2"].shape -> torch.Size([1, 256, 200, 320]) # stride = 4
output["p3"].shape -> torch.Size([1, 256, 100, 160]) # stride = 8
output["p4"].shape -> torch.Size([1, 256, 50, 80]) # stride = 16
output["p5"].shape -> torch.Size([1, 256, 25, 40]) # stride = 32
output["p6"].shape -> torch.Size([1, 256, 13, 20]) # stride = 64
```

Each tensor size stands for (batch, channels, height, width). We use the feature dimensions above throughout the blog series. The P2-P5 features are fed to the box head and P6 is not used.

b) **proposal boxes** are included in the output instances from RPN (see Part 4) which have 1000 'proposal_boxes' and 1000 'objectness_logits'. In the ROI Heads, only proposal boxes are used to crop the feature map and deal with the ROIs and objectness_logits is not used.

```
{'proposal_boxes':
    Boxes(tensor([[675.1985, 469.0636, 936.3209, 695.8753],
                  [301.7026, 513.4204, 324.4264, 572.4883],
                  [314.1965, 448.9897, 381.7842, 491.7808],
                ...,
'objectness_logits':
    tensor([ 9.1980,  8.0897,  8.0897, ...]
}
```

c) **ground truth boxes** have been loaded from the dataset (see Part 3):

```
'gt_boxes': Boxes(tensor([
[100.55, 180.24, 114.63, 103.01],
[180.58, 162.66, 204.78, 180.95]
])),
'gt_classes': tensor([9, 9])
```

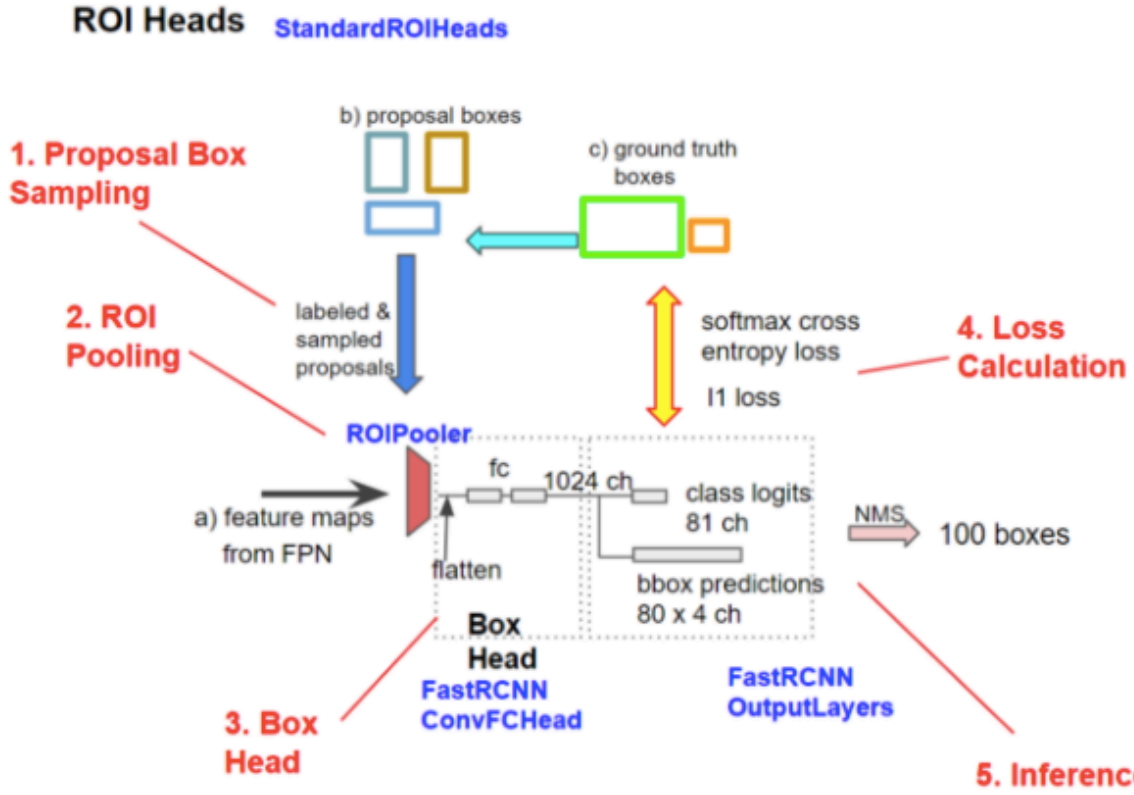


Fig 20. Schematic of ROI Heads. Blue and red labels represent class names and chapter titles respectively.

Fig 20 shows the detailed schematic of the ROI Heads. All the computation is performed on GPU in Detectron 2.

1. Proposal Box Sampling

(only during training)

In RPN, we have obtained 1,000 proposal boxes from the five levels of FPN features (P2 to P6).

The proposal boxes are used to crop the regions of interest (ROIs) from the feature maps, which are fed to the Box Head. To accelerate the training, ground-truth boxes are added to the predicted proposals. For example, if the image has two ground truth boxes, the total number of proposals will be 1002.

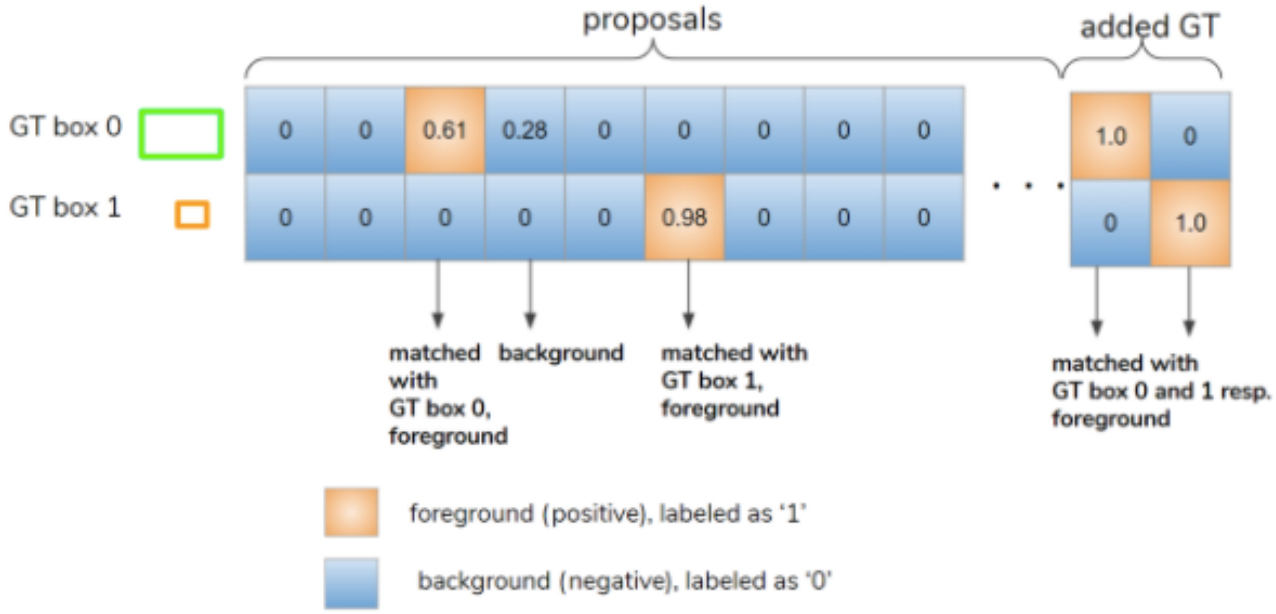


Fig 21. Matcher determines assignment of anchors to ground-truth boxes. The table shows the IoU matrix whose shape is (number of GT boxes, number of anchors).

During training, the foreground and background proposal boxes are firstly re-sampled to balance the training objective.

The proposals that have higher IoUs than the threshold are counted as foreground and the others as background by using Matcher (see Fig 21). Note that in ROI Heads there are no 'ignored' boxes unlike RPN. Added ground-truth boxes perfectly match themselves, thus are counted as foreground.

Next, we balance the numbers of foreground and background boxes. Let N be the target number of (foreground + background) boxes and F be the target number of foreground boxes. N and F / N are defined by the following config parameters.

The boxes are sampled as shown in Fig 22 so that the number of foreground boxes is less than F.

`N: MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE (typically 512)`
`F/N: MODEL.ROI_HEADS.POSITIVE_FRACTION (typically 0.25)`

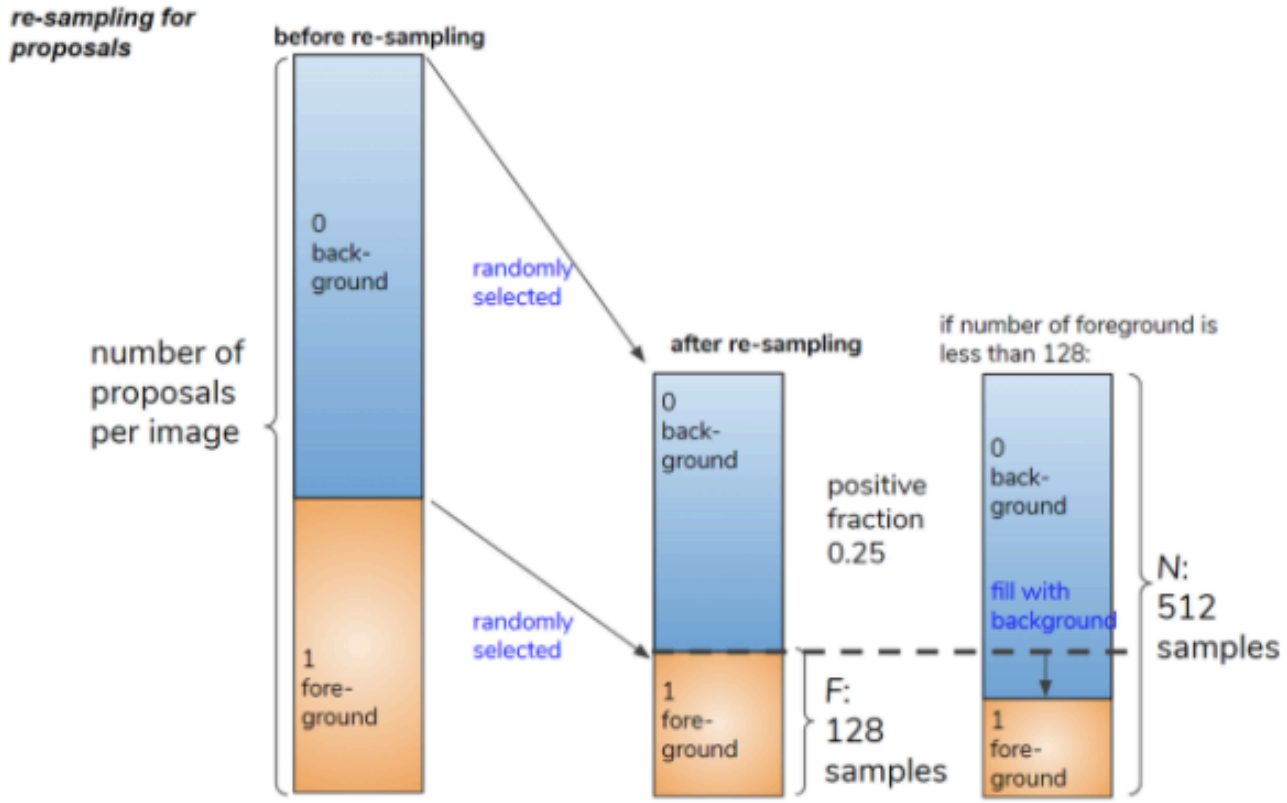


Figure 22. Re-sampling the foreground and background proposal boxes.

2. ROI Pooling

The ROI pooling process crops (or pools) the rectangle regions of the feature maps that are specified by the proposal boxes.

level assignment
Assume that we have two proposal boxes (gray and blue rectangles in Fig. 6) and the feature maps P2 to P5.

Which feature map should each box crop an ROI from? If you assign the small gray box to the P5 feature, only one or two feature pixels would be contained inside the box, which is not informative.

There is a rule to assign a proposal box to the appropriate feature map:
assigned feature level: floor(4 + log2(sqrt(box_area) / 224))
where 224 is the **canonical** box size. For example, if the size of the proposal box is 224×224, it is assigned to the 4th level (P4).

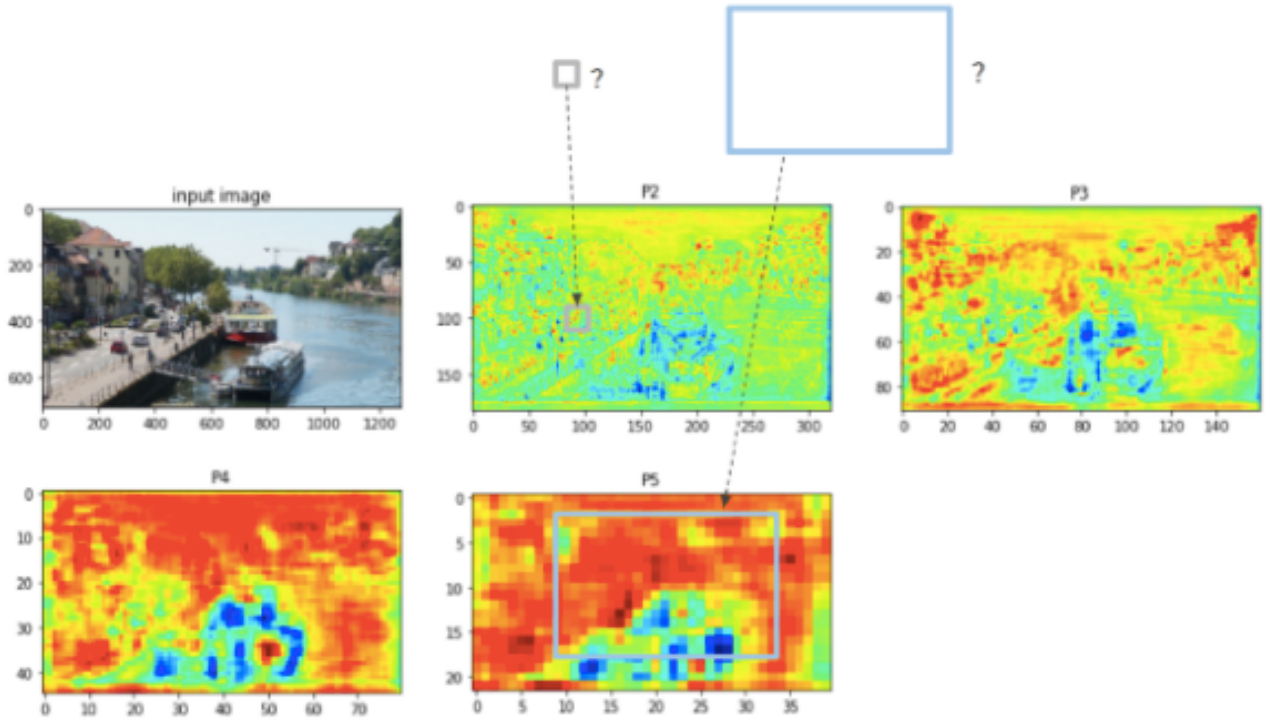


Fig 23. feature level assignment of proposal boxes for ROI pooling.

In case of Fig 23, the gray box is assigned to the P2 level and the blue one to the P5. The level assignment is carried out at the assign_boxes_to_levels function.

ROIAlignV2

In order to accurately crop the ROI by the proposal boxes which have floating-point coordinates, a method called ROIAlign has been proposed in the Mask R-CNN paper[4]. In Detectron 2, the default pooling method is called ROIAlignV2, which is the slightly modified version of ROIAlign.

In Fig 24, both ROIAlignV2 and ROIAlign are depicted. A large rectangle is one bin (or pixel) in an ROI. To pool the feature value inside the rectangle, four sampling points are placed to interpolate the four neighboring pixel values. The final bin value is calculated by averaging the four sampling point values. The difference between ROIAlignV2 and ROIAlign is simple. The half-pixel offset (0.5) is subtracted from ROI coordinates to compute neighboring pixel indices more accurately. Please look at Fig 24 for the details.

Now the ROIs are cropped from the corresponding levels (P2-P5) by ROIAlignV2. The resulting tensor has the size of:
[B, C, H, W] = [N × batch size, 256, 7, 7]

where B, C, H and W stand for the number of ROIs across the batch, channel number, height and width respectively. By default the number of ROIs for one batch

N is 512 and the ROI size 7 ×7. The tensor is the collection of cropped instance features which include balanced foreground and background ROIs.

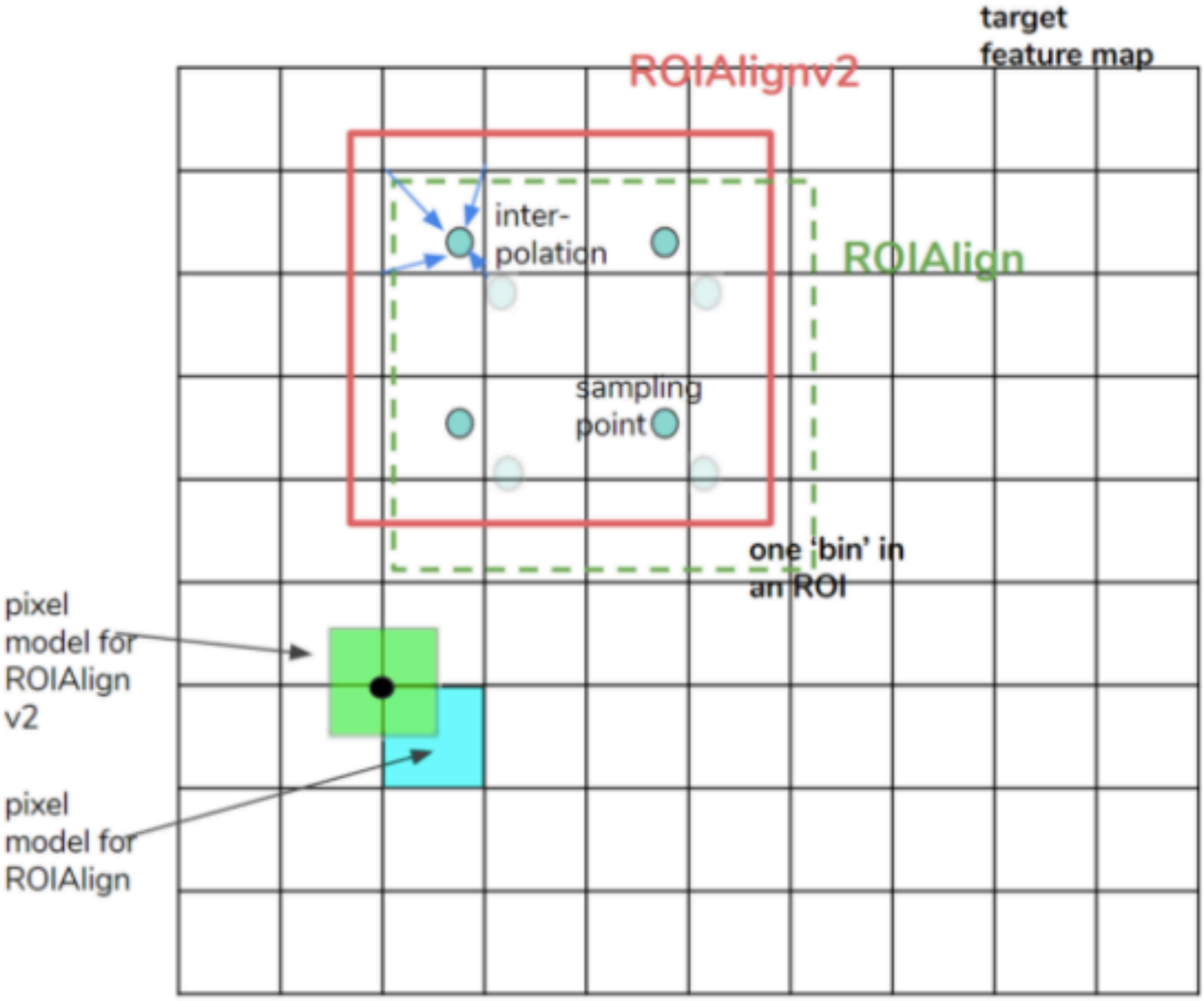


Fig 24. ROIAlignv2. Compared with ROIAlign(v1), the half-pixel offset (0.5) is subtracted from ROI coordinates to compute neighboring pixel indices more accurately. ROIAlignV2 employs the pixel model in which pixel coordinates represent the centers of pixels.

3. Box Head

After ROI Pooling, the cropped features are fed to the head networks. As for Mask R-CNN[4], there are two types of heads: the box head and the mask head. However Base R-CNN FPN only has the box head called FastRCNNConvFCHead which classifies the object within the ROI and fine-tunes the box position and shape.
The layers of the Box Head by default are as follows:

```
(box_head): FastRCNNConvFCHead(
      (fc1): Linear(in_features=12544, out_features=1024, bias=True)
      (fc2): Linear(in_features=1024, out_features=1024, bias=True)
    )
    (box_predictor): FastRCNNOutputLayers(
      (cls_score): Linear(in_features=1024, out_features=81, bias=True)
      (bbox_pred): Linear(in_features=1024, out_features=320, bias=True)
```

As you can see, no convolution layers are included in the head.
The input tensor whose size is [B, 256, 7, 7] is flattened to [B, 256×7 ×7 = 12,544 channels] to be fed to the fully-connected (FC) layer 1 (fc1).

After two FC layers the tensor gets to the final box_predictor layers: cls_score (Linear) and bbox_pred (Linear).

The output tensors from the final layers are:

```
cls_score -> scores             # shape:(B, 80+1)
bbox_pred -> prediction_deltas  # shape:(B, 80×4)
```

Next we see how to calculate the loss for the outputs during training.

4. Loss Calculation

(only during training)

Two loss functions are applied to the final output tensors.

localization loss (loss_box_reg)

- l1 loss[5].
- **foreground predictions** are picked from the *pred_proposal_deltas* tensor whose shape is (N samples × batch size, 80×4). For example, if the 15th sample is a foreground with class index = 17, the indices of [14 (=15–1), [68 (=17×4), 69, 70, 71]] are selected.
- **foreground ground truth targets** are picked from *gt_proposal_deltas* whose shape is (B, 4). The tensor values are the relative sizes of the ground truth boxes compared with the proposal boxes, which are calculated by the Box2BoxTransform.get_deltas function (see section 3–3 of Part4). The tensor with foreground indices are sampled from gt_proposal_deltas.

classification loss (loss_cls)

- Softmax cross entropy loss.
- Calculated for all the foreground and background prediction scores [B, K classes] vs ground truth class index [B]
- Classification objectives are **both foreground and background classes**, so K = number of classes + 1 (background class index is '80' for COCO dataset).

The loss results below are added to the losses calculated in RPN — 'loss_rpn_cls' and 'loss_rpn_cls' — and summed up to be the pipeline's total loss.

```
{
'loss_cls': tensor(4.3722, device='cuda:0',
grad_fn=<NllLossBackward>),
 'loss_box_reg': tensor(0.0533, device='cuda:0',
grad_fn=<DivBackward0>)
}
```

5. Inference

(only during test)

As we saw in Section 3, we have *scores* whose shape is (B, 80+1) and *prediction_deltas* whose shape is (B, 80×4) as output from the Box Head.

1. apply prediction deltas to proposal boxes

To calculate the final box coordinates from the prediction deltas[6] : Δx, Δy, Δw, and Δh, Box2BoxTransform.apply_deltas function is used (Fig 25). This is the same procedure as the step 1 in the section 5 of Part 4.

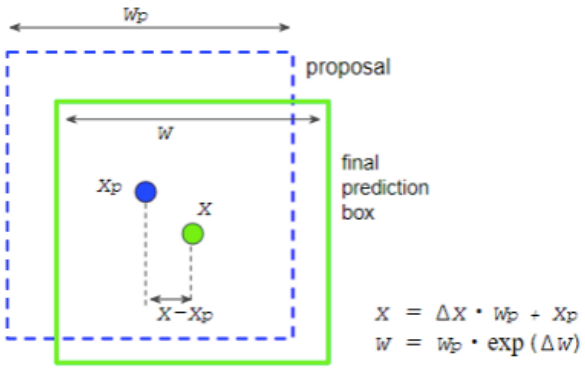


Fig 25. applying prediction deltas to a proposal box to calculate the coordinates of the final prediction box.

2. filter the boxes by scores

We firstly filter out the low-scored bounding boxes as shown in Fig. 26 (left to center). Each box has a corresponding score, so it's quite easy to do that.

Fig 26. Post-processing at the inference stage. left: visualization of all the ROIs before post-processing. center: after score thresholding. right: after non-maximum suppression.

3. non-maximum suppression

To remove the overlapping boxes, non-maximum suppression (NMS) is applied (Fig. 9, center to right). The parameter of NMS is defined here.

4. choose top-k results

Lastly we choose the top-k results when the number of the remaining boxes is more than the pre-defined number.