setup config

detectron2 中 类通过 @configurable **init**和 @classmethod from_config 方法实现直接从 config 提取对应的类初始化参数。

```
 def __init__(
        self,
        *,
        input_shape: List[ShapeSpec],
        num_classes,
        num_anchors,
        conv_dims: List[int],
        norm="",
        prior_prob=0.01,
    ):
        """
        NOTE: this interface is experimental.

        Args:
            input_shape (List[ShapeSpec]): input shape
            num_classes (int): number of classes. Used to label background
proposals.
            num_anchors (int): number of generated anchors
            conv_dims (List[int]): dimensions for each convolution layer
            norm (str or callable):
                    Normalization for conv layers except for the two output
layers.
                    See :func:`detectron2.layers.get_norm` for supported types.
            prior_prob (float): Prior weight for computing bias
        """
        super().__init__()


@classmethod
def from_config(cls, cfg, input_shape: List[ShapeSpec]):
    num_anchors = build_anchor_generator(cfg, input_shape).num_cell_anchors
    assert (
        len(set(num_anchors)) == 1
    ), "Using different number of anchors between levels is not currently
supported!"
    num_anchors = num_anchors[0]

    return {
        "input_shape": input_shape,
        "num_classes": cfg.MODEL.RETINANET.NUM_CLASSES,
        "conv_dims": [input_shape[0].channels] * cfg.MODEL.RETINANET.NUM_CONVS,
        "prior_prob": cfg.MODEL.RETINANET.PRIOR_PROB,
```

```
        "norm": cfg.MODEL.RETINANET.NORM,
        "num_anchors": num_anchors,
    }
```

1. anchor generator sizes

```
ANCHOR_GENERATOR:
    SIZES: !!python/object/apply:eval ["[[x, x * 2**(1.0/3), x * 2**(2.0/3)
] for x in [32, 64, 128, 256, 512 ]]"]
```

2. _C.MODEL.RETINANET.NUM_CONVS = 4: 控制 RetinaHead 中卷积层的个数

```
for in_channels, out_channels in zip([input_shape[0].channels] + conv_dims,
conv_dims):
    cls_subnet.append(
        nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1,
padding=1)
    )
    if norm:
        cls_subnet.append(get_norm(norm, out_channels))
    cls_subnet.append(nn.ReLU())
    bbox_subnet.append(
        nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1,
padding=1)
    )
    if norm:
        bbox_subnet.append(get_norm(norm, out_channels))
    bbox_subnet.append(nn.ReLU())
```

3. _C.MODEL.RETINANET.PRIOR_PROB = 0.01: 解决初始训练正负样本数目不均衡导致的梯度爆炸问题，让训练更加稳定。

```
# Use prior in model initialization to improve stability
bias_value = -(math.log((1 - prior_prob) / prior_prob))
torch.nn.init.constant_(self.cls_score.bias, bias_value)
```

If we set the prior_prob to high, there will be loss explosion error:

```
 f"Loss became infinite or NaN at iteration={self.iter}!\n"
FloatingPointError: Loss became infinite or NaN at iteration=42!
loss_dict = {'loss_cls': nan, 'loss_box_reg': nan}
```

setup training

构建模型，优化器，数据加载

```
model = self.build_model(cfg)
optimizer = self.build_optimizer(cfg, model)
data_loader = self.build_train_loader(cfg)
```

build model

1. build retinanet backbone

   使用 resnet 提取基础特征，然后使用 FPN 对特征进行增强，并在 res3, res4, res5 基础上增加两个 stride 2 卷积，进一步对 feature map 进行下采样

   ```
   bottom_up = build_resnet_backbone(cfg, input_shape)
   in_features = cfg.MODEL.FPN.IN_FEATURES
   out_channels = cfg.MODEL.FPN.OUT_CHANNELS
   in_channels_p6p7 = bottom_up.output_shape()["res5"].channels
   backbone = FPN(
       bottom_up=bottom_up,
       in_features=in_features,
       out_channels=out_channels,
       norm=cfg.MODEL.FPN.NORM,
       top_block=LastLevelP6P7(in_channels_p6p7, out_channels),
       fuse_type=cfg.MODEL.FPN.FUSE_TYPE,
   )
   ```

   LastLevelP6P7: 对特征进一步下采样

   ```
   class LastLevelP6P7(nn.Module):
    """
    This module is used in RetinaNet to generate extra layers, P6 and P7 from
    C5 feature.
    """

    def __init__(self, in_channels, out_channels, in_feature="res5"):
        super().__init__()
        self.num_levels = 2
        self.in_feature = in_feature
        self.p6 = nn.Conv2d(in_channels, out_channels, 3, 2, 1)
     self.p7 = nn.Conv2d(out_channels, out_channels, 3, 2, 1)
        for module in [self.p6, self.p7]:
            weight_init.c2_xavier_fill(module)

    def forward(self, c5):
        p6 = self.p6(c5)
        p7 = self.p7(F.relu(p6))
        return [p6, p7]
   ```

2. build retinanet head retinanet head 在增强的特征基础上进行 anchor 的分类和回归

```
 cls_subnet = []
     bbox_subnet = []
     for in_channels, out_channels in zip([input_shape[0].channels] +
conv_dims, conv_dims):
         cls_subnet.append(
             nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1,
padding=1)
         )
         if norm:
             cls_subnet.append(get_norm(norm, out_channels))
         cls_subnet.append(nn.ReLU())
         bbox_subnet.append(
             nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1,
padding=1)
         )
         if norm:
             bbox_subnet.append(get_norm(norm, out_channels))
         bbox_subnet.append(nn.ReLU())

     self.cls_subnet = nn.Sequential(*cls_subnet)
     self.bbox_subnet = nn.Sequential(*bbox_subnet)
     self.cls_score = nn.Conv2d(
         conv_dims[-1], num_anchors * num_classes, kernel_size=3, stride=1,
padding=1
     )
     self.bbox_pred = nn.Conv2d(
         conv_dims[-1], num_anchors * 4, kernel_size=3, stride=1, padding=1
     )
```

build optimizer

构建优化器，默认是SGD

```
def build_optimizer(cfg: CfgNode, model: torch.nn.Module) ->
torch.optim.Optimizer:
    """
    Build an optimizer from config.
    """
    params = get_default_optimizer_params(
        model,
        base_lr=cfg.SOLVER.BASE_LR,
        weight_decay=cfg.SOLVER.WEIGHT_DECAY,
        weight_decay_norm=cfg.SOLVER.WEIGHT_DECAY_NORM,
        bias_lr_factor=cfg.SOLVER.BIAS_LR_FACTOR,
        weight_decay_bias=cfg.SOLVER.WEIGHT_DECAY_BIAS,
    )
    return maybe_add_gradient_clipping(cfg, torch.optim.SGD)(
```

```
            params, cfg.SOLVER.BASE_LR, momentum=cfg.SOLVER.MOMENTUM,
nesterov=cfg.SOLVER.NESTEROV
    )
```

build data loader

data loader 中读取的数据是轻量结构，此时图片没有从磁盘中读入。需要使用DatasetMapper读入图像并进行一系列增强处理。

```
# dataset_mapper.py


dataset_dict = copy.deepcopy(dataset_dict)  # it will be modified by code below
# USER: Write your own image loading if it's not from a file
image = utils.read_image(dataset_dict["file_name"], format=self.image_format)
utils.check_image_size(dataset_dict, image)

aug_input = T.AugInput(image, sem_seg=sem_seg_gt)
transforms = self.augmentations(aug_input)
image, sem_seg_gt = aug_input.image, aug_input.sem_seg
```

增强的配置： Trainer -> build_detection_train_loader(cfg) -> @configurable(from_config=_train_loader_from_config) 使用装饰器从 config 传入参数，实现动态多尺度训练和水平翻转增强。

```
def build_augmentation(cfg, is_train):
    """
    Create a list of default :class:`Augmentation` from config.
    Now it includes resizing and flipping.

    Returns:
        list[Augmentation]
    """
    if is_train:
        min_size = cfg.INPUT.MIN_SIZE_TRAIN
        max_size = cfg.INPUT.MAX_SIZE_TRAIN
        sample_style = cfg.INPUT.MIN_SIZE_TRAIN_SAMPLING
    else:
        min_size = cfg.INPUT.MIN_SIZE_TEST
        max_size = cfg.INPUT.MAX_SIZE_TEST
        sample_style = "choice"
    augmentation = [T.ResizeShortestEdge(min_size, max_size, sample_style)]
    if is_train and cfg.INPUT.RANDOM_FLIP != "none":
        augmentation.append(
            T.RandomFlip(
                horizontal=cfg.INPUT.RANDOM_FLIP == "horizontal",
                vertical=cfg.INPUT.RANDOM_FLIP == "vertical",
            )
```

```
        )
    return augmentation
```

build lr_scheduler

构建学习率调节器

```
def build_lr_scheduler(
    cfg: CfgNode, optimizer: torch.optim.Optimizer
) -> torch.optim.lr_scheduler._LRScheduler:
    """
    Build a LR scheduler from config.
    """
    name = cfg.SOLVER.LR_SCHEDULER_NAME
    if name == "WarmupMultiStepLR":
        return WarmupMultiStepLR(
            optimizer,
            cfg.SOLVER.STEPS,
            cfg.SOLVER.GAMMA,
            warmup_factor=cfg.SOLVER.WARMUP_FACTOR,
            warmup_iters=cfg.SOLVER.WARMUP_ITERS,
            warmup_method=cfg.SOLVER.WARMUP_METHOD,
        )
    elif name == "WarmupCosineLR":
        return WarmupCosineLR(
            optimizer,
            cfg.SOLVER.MAX_ITER,
            warmup_factor=cfg.SOLVER.WARMUP_FACTOR,
            warmup_iters=cfg.SOLVER.WARMUP_ITERS,
            warmup_method=cfg.SOLVER.WARMUP_METHOD,
        )
    else:
        raise ValueError("Unknown LR scheduler: {}".format(name))
```

WarmupMultiStepLR

```
 def get_lr(self) -> List[float]:
     warmup_factor = _get_warmup_factor_at_iter(
         self.warmup_method, self.last_epoch, self.warmup_iters,
self.warmup_factor
     )
     return [
         base_lr * warmup_factor * self.gamma ** bisect_right(self.milestones,
self.last_epoch)
         for base_lr in self.base_lrs
     ]
```

training

process image

归一化，并对图像进行 pad, 满足 下采样条件。

```
 def preprocess_image(self, batched_inputs: Tuple[Dict[str, Tensor]]):
        """
        Normalize, pad and batch the input images.
        """
        images = [x["image"].to(self.device) for x in batched_inputs]
        images = [(x - self.pixel_mean) / self.pixel_std for x in images]
        images = ImageList.from_tensors(images,
self.backbone.size_divisibility)
        return images
```

backbone feature from resnet and fpn

提取经过 resnet 和 fpn 增强的特征金字塔

```
features = self.backbone(images.tensor)
```

generate anchors

generate anchors according to feature map size, anchor size, and anchor aspect ratio.

```
# anchor_generator.py
grid_sizes = [feature_map.shape[-2:] for feature_map in features]
anchors_over_all_feature_maps = self._grid_anchors(grid_sizes)
return [RotatedBoxes(x) for x in anchors_over_all_feature_maps]
```

predict logits and box delta using retinanet head

```
  logits = []
  bbox_reg = []
  for feature in features:
    logits.append(self.cls_score(self.cls_subnet(feature)))
    bbox_reg.append(self.bbox_pred(self.bbox_subnet(feature)))
    return logits, bbox_reg
```

match anchor with ground truth

将 anchor 和 ground truth 进行匹配，从而生成每个 anchor 训练的时候需要的类别标签和回归目标。

```
gt_labels, gt_boxes = self.label_anchors(anchors, gt_instances)
```

```
 def label_anchors(self, anchors, gt_instances):
     """
     Args:
         anchors (list[Boxes]): A list of #feature level Boxes.
             The Boxes contains anchors of this image on the specific feature
level.
         gt_instances (list[Instances]): a list of N `Instances`s. The i-th
             `Instances` contains the ground-truth per-instance annotations
             for the i-th input image.

     Returns:
         list[Tensor]:
             List of #img tensors. i-th element is a vector of labels whose
length is
             the total number of anchors across all feature maps (sum(Hi * Wi *
A)).
             Label values are in {-1, 0, ..., K}, with -1 means ignore, and K
means background.
         list[Tensor]:
             i-th element is a Rx4 tensor, where R is the total number of
anchors across
             feature maps. The values are the matched gt boxes for each anchor.
             Values are undefined for those anchors not labeled as foreground.
     """
     anchors = Boxes.cat(anchors)  # Rx4

     gt_labels = []
     matched_gt_boxes = []
     for gt_per_image in gt_instances:
         match_quality_matrix = pairwise_iou(gt_per_image.gt_boxes, anchors)
         matched_idxs, anchor_labels =
self.anchor_matcher(match_quality_matrix)
         del match_quality_matrix

         if len(gt_per_image) > 0:
             matched_gt_boxes_i = gt_per_image.gt_boxes.tensor[matched_idxs]

             gt_labels_i = gt_per_image.gt_classes[matched_idxs]
             # Anchors with label 0 are treated as background.
             gt_labels_i[anchor_labels == 0] = self.num_classes
             # Anchors with label -1 are ignored.
             gt_labels_i[anchor_labels == -1] = -1
         else:
             matched_gt_boxes_i = torch.zeros_like(anchors.tensor)
             gt_labels_i = torch.zeros_like(matched_idxs) + self.num_classes

         gt_labels.append(gt_labels_i)
         matched_gt_boxes.append(matched_gt_boxes_i)
```

```
    return gt_labels, matched_gt_boxes
```

calculate loss

计算损失函数，根据 retinanet head 预测的类别和标签结果和前面匹配的真实类别和回归目标分别计算 focal loss 和 smooth l1 loss

```
 def losses(self, anchors, pred_logits, gt_labels, pred_anchor_deltas,
gt_boxes):
        """
        Args:
            anchors (list[Boxes]): a list of #feature level Boxes
            gt_labels, gt_boxes: see output of :meth:`RetinaNet.label_anchors`.
                Their shapes are (N, R) and (N, R, 4), respectively, where R is
                the total number of anchors across levels, i.e. sum(Hi x Wi x
Ai)
            pred_logits, pred_anchor_deltas: both are list[Tensor]. Each
element in the
                list corresponds to one level and has shape (N, Hi * Wi * Ai, K
or 4).
                Where K is the number of classes used in `pred_logits`.

        Returns:
            dict[str, Tensor]:
                mapping from a named loss to a scalar tensor
                storing the loss. Used during training only. The dict keys are:
                "loss_cls" and "loss_box_reg"
        """
        num_images = len(gt_labels)
        gt_labels = torch.stack(gt_labels)  # (N, R)
        anchors = type(anchors[0]).cat(anchors).tensor  # (R, 4)
        gt_anchor_deltas = [self.box2box_transform.get_deltas(anchors, k) for k
in gt_boxes]
        gt_anchor_deltas = torch.stack(gt_anchor_deltas)  # (N, R, 4)

        valid_mask = gt_labels >= 0
        pos_mask = (gt_labels >= 0) & (gt_labels != self.num_classes)
        num_pos_anchors = pos_mask.sum().item()
        get_event_storage().put_scalar("num_pos_anchors", num_pos_anchors /
num_images)
        self.loss_normalizer = self.loss_normalizer_momentum *
self.loss_normalizer + (
            1 - self.loss_normalizer_momentum
        ) * max(num_pos_anchors, 1)

        # classification and regression loss
```

```
        gt_labels_target = F.one_hot(gt_labels[valid_mask],
num_classes=self.num_classes + 1)[
            :, :-1
        ]  # no loss for the last (background) class
        loss_cls = sigmoid_focal_loss_jit(
            cat(pred_logits, dim=1)[valid_mask],
            gt_labels_target.to(pred_logits[0].dtype),
            alpha=self.focal_loss_alpha,
            gamma=self.focal_loss_gamma,
            reduction="sum",
        )

        if self.box_reg_loss_type == "smooth_l1":
            loss_box_reg = smooth_l1_loss(
                cat(pred_anchor_deltas, dim=1)[pos_mask],
                gt_anchor_deltas[pos_mask],
                beta=self.smooth_l1_beta,
                reduction="sum",
            )
        elif self.box_reg_loss_type == "giou":
            pred_boxes = [
                self.box2box_transform.apply_deltas(k, anchors)
                for k in cat(pred_anchor_deltas, dim=1)
            ]
            loss_box_reg = giou_loss(
                torch.stack(pred_boxes)[pos_mask], torch.stack(gt_boxes)
[pos_mask], reduction="sum"
            )
        else:
            raise ValueError(f"Invalid bbox reg loss type
'{self.box_reg_loss_type}'")

        return {
            "loss_cls": loss_cls / self.loss_normalizer,
            "loss_box_reg": loss_box_reg / self.loss_normalizer,
        }
```