two_stage_detector

September 7, 2024

[ ]:
```
import math
from typing import Dict, List, Optional, Tuple
```

[ ]:
```
import torch
import torchvision
from a4_helper import *
from common import class_spec_nms, get_fpn_location_coords, nms
from torch import nn
from torch.nn import functional as F
```

[ ]:
```
# Short hand type notation:
TensorDict = Dict[str, torch.Tensor]
```

[ ]:
```
def hello_two_stage_detector():
    print("Hello from two_stage_detector.py!")
```

[ ]:
```
class RPNPredictionNetwork(nn.Module):
    """
    RPN prediction network that accepts FPN feature maps from different levels
    and makes two predictions for every anchor: objectness and box deltas.

    Faster R-CNN typically uses (p2, p3, p4, p5) feature maps. We will exclude
    p2 for have a small enough model for Colab.

    Conceptually this module is quite similar to `FCOSPredictionNetwork`.
    """

    def __init__(
        self, in_channels: int, stem_channels: List[int], num_anchors: int = 3
    ):
        """
        Args:
            in_channels: Number of channels in input feature maps. This value
                is same as the output channels of FPN.
            stem_channels: List of integers giving the number of output channels
                in each convolution layer of stem layers.
            num_anchors: Number of anchor boxes assumed per location (say, `A`).
                Faster R-CNN without an FPN uses `A = 9`, anchors with three
                different sizes and aspect ratios. With FPN, it is more common
                to have a fixed size dependent on the stride of FPN level, hence
                `A = 3` is default - with three aspect ratios.
        """
        super().__init__()

        self.num_anchors = num_anchors
        ######################################################################
        # TODO: Create a stem of alternating 3x3 convolution layers and RELU
        # activation modules. RPN shares this stem for objectness and box
        # regression (unlike FCOS, that uses separate stems).
        #
        # Use `in_channels` and `stem_channels` for creating these layers, the
        # docstring above tells you what they mean. Initialize weights of each
        # conv layer from a normal distribution with mean = 0 and std dev = 0.01
        # and all biases with zero. Use conv stride = 1 and zero padding such
        # that size of input features remains same: remember we need predictions
        # at every location in feature map, we shouldn't "lose" any locations.
        ######################################################################
        # Fill this list. It is okay to use your implementation from
        # `FCOSPredictionNetwork` for this code block.
        stem_rpn = []
        # Replace "pass" statement with your code
        pass

        # Wrap the layers defined by student into a `nn.Sequential` module:
        self.stem_rpn = nn.Sequential(*stem_rpn)
        ######################################################################
        # TODO: Create TWO 1x1 conv layers for individually to predict
        # objectness and box deltas for every anchor, at every location.
        #
        # Objectness is obtained by applying sigmoid to its logits. However,
        # DO NOT initialize a sigmoid module here. PyTorch loss functions have
        # numerically stable implementations with logits.
        ######################################################################

        # Replace these lines with your code, keep variable names unchanged.
        self.pred_obj = None  # Objectness conv
        self.pred_box = None  # Box regression conv

        # Replace "pass" statement with your code
        pass
        ######################################################################
        #                             END OF YOUR CODE                       #
        ######################################################################

    def forward(self, feats_per_fpn_level: TensorDict) -> List[TensorDict]:
        """
        Accept FPN feature maps and predict desired quantities for every anchor
        at every location. Format the output tensors such that feature height,
        width, and number of anchors are collapsed into a single dimension (see
        description below in "Returns" section) this is convenient for computing
        loss and perforning inference.

        Args:
            feats_per_fpn_level: Features from FPN, keys {"p3", "p4", "p5"}.
                Each tensor will have shape `(batch_size, fpn_channels, H, W)`.

        Returns:
            List of dictionaries, each having keys {"p3", "p4", "p5"}:
            1. Objectness logits:     `(batch_size, H * W * num_anchors)`
            2. Box regression deltas: `(batch_size, H * W * num_anchors, 4)`
        """

        ######################################################################
        # TODO: Iterate over every FPN feature map and obtain predictions using
        # the layers defined above. DO NOT apply sigmoid to objectness logits.
        ######################################################################
        # Fill these with keys: {"p3", "p4", "p5"}, same as input dictionary.
        object_logits = {}
        boxreg_deltas = {}

        # Replace "pass" statement with your code
        pass
        ######################################################################
        #                           END OF YOUR CODE                         #
        ######################################################################

        return [object_logits, boxreg_deltas]
```

[ ]:
```
@torch.no_grad()
def generate_fpn_anchors(
    locations_per_fpn_level: TensorDict,
    strides_per_fpn_level: Dict[str, int],
    stride_scale: int,
    aspect_ratios: List[float] = [0.5, 1.0, 2.0],
):
    """
    Generate multiple anchor boxes at every location of FPN level. Anchor boxes
    should be in XYXY format and they should be centered at the given locations.

    Args:
        locations_per_fpn_level: Centers at different levels of FPN (p3, p4,␣
 ↪p5),
            that are already projected to absolute co-ordinates in input image
            dimension. Dictionary of three keys: (p3, p4, p5) giving tensors of
            shape `(H * W, 2)` where H, W is the size of FPN feature map.
        strides_per_fpn_level: Dictionary of same keys as above, each with an
            integer value giving the stride of corresponding FPN level.
            See `common.py` for more details.
        stride_scale: Size of square anchor at every FPN levels will be
            `(this value) * (FPN level stride)`. Default is 4, which will make
            anchor boxes of size (32x32), (64x64), (128x128) for FPN levels
            p3, p4, and p5 respectively.
        aspect_ratios: Anchor aspect ratios to consider at every location. We
            consider anchor area to be `(stride_scale * FPN level stride) ** 2`
            and set new width and height of anchors at every location:
                new_width = sqrt(area / aspect ratio)
                new_height = area / new_width

    Returns:
        TensorDict
            Dictionary with same keys as `locations_per_fpn_level` and values as
            tensors of shape `(HWA, 4)` giving anchors for all locations
            per FPN level, each location having `A = len(aspect_ratios)`␣
↪anchors.
            All anchors are in XYXY format and their centers align with␣
↪locations.
    """

    # Set these to `(N, A, 4)` Tensors giving anchor boxes in XYXY format.
    anchors_per_fpn_level = {
        level_name: None for level_name, _ in locations_per_fpn_level.items()
    }

    for level_name, locations in locations_per_fpn_level.items():
        level_stride = strides_per_fpn_level[level_name]

        # List of `A = len(aspect_ratios)` anchor boxes.
        anchor_boxes = []
        for aspect_ratio in aspect_ratios:
            ##############################################################
            # TODO: Implement logic for anchor boxes below. Write vectorized
            # implementation to generate anchors for a single aspect ratio.
            # Fill `anchor_boxes` list above.
            #
            # Calculate resulting width and height of the anchor box as per
            # `stride_scale` and `aspect_ratios` definitions. Then shift the
            # locations to get top-left and bottom-right co-ordinates.
            ##############################################################
            # Replace "pass" statement with your code
            pass
            ####################################################################
            #                          END OF YOUR CODE                        #
            ####################################################################

        # shape: (A, H * W, 4)
        anchor_boxes = torch.stack(anchor_boxes)
        # Bring `H * W` first and collapse those dimensions.
        anchor_boxes = anchor_boxes.permute(1, 0, 2).contiguous().view(-1, 4)
        anchors_per_fpn_level[level_name] = anchor_boxes

    return anchors_per_fpn_level
```

[ ]:
```
@torch.no_grad()
def iou(boxes1: torch.Tensor, boxes2: torch.Tensor) -> torch.Tensor:
    """
    Compute intersection-over-union (IoU) between pairs of box tensors. Input
    box tensors must in XYXY format.

    Args:
        boxes1: Tensor of shape `(M, 4)` giving a set of box co-ordinates.
        boxes2: Tensor of shape `(N, 4)` giving another set of box co-ordinates.

    Returns:
        torch.Tensor
            Tensor of shape (M, N) with `iou[i, j]` giving IoU between i-th box
            in `boxes1` and j-th box in `boxes2`.
    """

    ##########################################################################
    # TODO: Implement the IoU function here.                                 #
    ##########################################################################
    # Replace "pass" statement with your code
    pass
    ##########################################################################
    #                             END OF YOUR CODE                           #
    ##########################################################################
    return iou
```

[ ]:
```
@torch.no_grad()
def rcnn_match_anchors_to_gt(
    anchor_boxes: torch.Tensor,
    gt_boxes: torch.Tensor,
    iou_thresholds: Tuple[float, float],
) -> TensorDict:
    """
    Match anchor boxes (or RPN proposals) with a set of GT boxes. Anchors having
    high IoU with any GT box are assigned "foreground" and matched with that box
    or vice-versa.

    NOTE: This function is NOT BATCHED. Call separately for GT boxes per image.

    Args:
        anchor_boxes: Anchor boxes (or RPN proposals). Dictionary of three keys
            a combined tensor of some shape `(N, 4)` where `N` are total anchors
            from all FPN levels, or a set of RPN proposals.
        gt_boxes: GT boxes of a single image, a batch of `(M, 5)` boxes with
            absolute co-ordinates and class ID `(x1, y1, x2, y2, C)`. In this
            codebase, this tensor is directly served by the dataloader.
        iou_thresholds: Tuple of (low, high) IoU thresholds, both in [0, 1]
            giving thresholds to assign foreground/background anchors.
    """

    # Filter empty GT boxes:
    gt_boxes = gt_boxes[gt_boxes[:, 4] != -1]

    # If no GT boxes are available, match all anchors to background and return.
    if len(gt_boxes) == 0:
        fake_boxes = torch.zeros_like(anchor_boxes) - 1
        fake_class = torch.zeros_like(anchor_boxes[:, [0]]) - 1
        return torch.cat([fake_boxes, fake_class], dim=1)

    # Match matrix => pairwise IoU of anchors (rows) and GT boxes (columns).
    # STUDENTS: This matching depends on your IoU implementation.
    match_matrix = iou(anchor_boxes, gt_boxes[:, :4])

    # Find matched ground-truth instance per anchor:
    match_quality, matched_idxs = match_matrix.max(dim=1)
    matched_gt_boxes = gt_boxes[matched_idxs]

    # Set boxes with low IoU threshold to background (-1).
    matched_gt_boxes[match_quality <= iou_thresholds[0]] = -1

    # Set remaining boxes to neutral (-1e8).
    neutral_idxs = (match_quality > iou_thresholds[0]) & (
        match_quality < iou_thresholds[1]
    )
    matched_gt_boxes[neutral_idxs, :] = -1e8
    return matched_gt_boxes
```

[ ]:
```
def rcnn_get_deltas_from_anchors(
    anchors: torch.Tensor, gt_boxes: torch.Tensor
) -> torch.Tensor:
    """
    Get box regression deltas that transform `anchors` to `gt_boxes`. These
    deltas will become GT targets for box regression. Unlike FCOS, the deltas
    are in `(dx, dy, dw, dh)` format that represent offsets to anchor centers
    and scaling factors for anchor size. Box regression is only supervised by
    foreground anchors. If GT boxes are "background/neutral", then deltas
    must be `(-1e8, -1e8, -1e8, -1e8)` (just some LARGE negative number).

    Follow Slide 68:
        https://web.eecs.umich.edu/~justincj/slides/eecs498/WI2022/
↪598_WI2022_lecture13.pdf

    Args:
        anchors: Tensor of shape `(N, 4)` giving anchors boxes in XYXY format.
        gt_boxes: Tensor of shape `(N, 4)` giving matching GT boxes.

    Returns:
        torch.Tensor
            Tensor of shape `(N, 4)` giving anchor deltas.
    """
    ##########################################################################
    # TODO: Implement the logic to get deltas.                               #
    # Remember to set the deltas of "background/neutral" GT boxes to -1e8    #
    ##########################################################################
    deltas = None
    # Replace "pass" statement with your code
    pass
    ##########################################################################
    #                             END OF YOUR CODE                           #
    ##########################################################################

    return deltas
```

[ ]:
```
def rcnn_apply_deltas_to_anchors(
    deltas: torch.Tensor, anchors: torch.Tensor
) -> torch.Tensor:
    """
    Implement the inverse of `rcnn_get_deltas_from_anchors` here.

    Args:
        deltas: Tensor of shape `(N, 4)` giving box regression deltas.
        anchors: Tensor of shape `(N, 4)` giving anchors to apply deltas on.

    Returns:
        torch.Tensor
            Same shape as deltas and locations, giving the resulting boxes in
            XYXY format.
    """
    # Clamp dw and dh such that they would transform a 8px box no larger than
    # 224px. This is necessary for numerical stability as we apply exponential.
    scale_clamp = math.log(224 / 8)
    deltas[:, 2] = torch.clamp(deltas[:, 2], max=scale_clamp)
    deltas[:, 3] = torch.clamp(deltas[:, 3], max=scale_clamp)

    ##########################################################################
    # TODO: Implement the transformation logic to get output boxes.         #
    ##########################################################################
    output_boxes = None
    # Replace "pass" statement with your code
    pass
    ##########################################################################
    #                             END OF YOUR CODE                           #
    ##########################################################################
    return output_boxes
```

```
[ ]: @torch.no_grad()
     def sample_rpn_training(
         gt_boxes: torch.Tensor, num_samples: int, fg_fraction: float
     ):
         """
         Return `num_samples` (or fewer, if not enough found) random pairs of anchors
         and GT boxes without exceeding `fg_fraction * num_samples` positives, and
         then try to fill the remaining slots with background anchors. We will ignore
         "neutral" anchors in this sampling as they are not used for training.

         Args:
             gt_boxes: Tensor of shape `(N, 5)` giving GT box co-ordinates that are
                 already matched with some anchor boxes (with GT class label at last
                 dimension). Label -1 means background and -1e8 means meutral.
             num_samples: Total anchor-GT pairs with label >= -1 to return.
             fg_fraction: The number of subsampled labels with values >= 0 is
                 `min(num_foreground, int(fg_fraction * num_samples))`. In other
                 words, if there are not enough fg, the sample is filled with
                 (duplicate) bg.

         Returns:
             fg_idx, bg_idx (Tensor):
                 1D vector of indices. The total length of both is `num_samples` or
                 fewer. Use these to index anchors, GT boxes, and model predictions.
         """
         foreground = (gt_boxes[:, 4] >= 0).nonzero().squeeze(1)
         background = (gt_boxes[:, 4] == -1).nonzero().squeeze(1)

         # Protect against not enough foreground examples.
```

```
    num_fg = min(int(num_samples * fg_fraction), foreground.numel())
    num_bg = num_samples - num_fg

    # Randomly select positive and negative examples.
    perm1 = torch.randperm(foreground.numel(), device=foreground.device)[:
 ↪num_fg]
    perm2 = torch.randperm(background.numel(), device=background.device)[:
 ↪num_bg]

    fg_idx = foreground[perm1]
    bg_idx = background[perm2]
    return fg_idx, bg_idx
```

[ ]:
```
@torch.no_grad()
def mix_gt_with_proposals(
    proposals_per_fpn_level: Dict[str, List[torch.Tensor]], gt_boxes: torch.
 ↪Tensor
):
    """
    At start of training, RPN proposals may be low quality. It's possible that
    very few of these have high IoU with GT boxes. This may stall or␣
 ↪de-stabilize
    training of second stage. This function mixes GT boxes with RPN proposals to
    improve training. Different GT boxes are mixed with proposals from different
    FPN levels according to assignment rule of FPN paper.

    Args:
        proposals_per_fpn_level: Dict of proposals per FPN level, per image in
            batch. These are same as outputs from `RPN.forward()` method.
        gt_boxes: Tensor of shape `(B, M, 4 or 5)` giving GT boxes per image in
            batch (with or without GT class label, doesn't matter).

    Returns:
        proposals_per_fpn_level: Same as input, but with GT boxes mixed in them.
    """

    # Mix ground-truth boxes for every example, per FPN level. There's no direct
    # way to vectorize this.
    for _idx, _gtb in enumerate(gt_boxes):

        # Filter empty GT boxes:
        _gtb = _gtb[_gtb[:, 4] != -1]
        if len(_gtb) == 0:
            continue

        # Compute FPN level assignments for each GT box. This follows Equation␣
 ↪(1)
```

```
        # of FPN paper (k0 = 5). `level_assn` has `(M, )` integers, one of␣
↪{3,4,5}
        _gt_area = (_gtb[:, 2] - _gtb[:, 0]) * (_gtb[:, 3] - _gtb[:, 1])
        level_assn = torch.floor(5 + torch.log2(torch.sqrt(_gt_area) / 224))
        level_assn = torch.clamp(level_assn, min=3, max=5).to(torch.int64)

        for level_name, _props in proposals_per_fpn_level.items():
            _prop = _props[_idx]

            # Get GT boxes of this image that match level scale, and append them
            # to proposals.
            _gt_boxes_fpn_subset = _gtb[level_assn == int(level_name[1])]
            if len(_gt_boxes_fpn_subset) > 0:
                proposals_per_fpn_level[level_name][_idx] = torch.cat(
                    # Remove class label since proposals don't have it:
                    [_prop, _gt_boxes_fpn_subset[:, :4]],
                    dim=0,
                )

    return proposals_per_fpn_level
```

```
[ ]: class RPN(nn.Module):
    """
    Region Proposal Network: First stage of Faster R-CNN detector.

    This class puts together everything you implemented so far. It accepts FPN
    features as input and uses `RPNPredictionNetwork` to predict objectness and
    box reg deltas. Computes proposal boxes for second stage (during both
    training and inference) and losses during training.
    """

    def __init__(
        self,
        fpn_channels: int,
        stem_channels: List[int],
        batch_size_per_image: int,
        anchor_stride_scale: int = 8,
        anchor_aspect_ratios: List[int] = [0.5, 1.0, 2.0],
        anchor_iou_thresholds: Tuple[int, int] = (0.3, 0.6),
        nms_thresh: float = 0.7,
        pre_nms_topk: int = 400,
        post_nms_topk: int = 100,
    ):
        """
        Args:
            batch_size_per_image: Anchors per image to sample for training.
            nms_thresh: IoU threshold for NMS - unlike FCOS, this is used
            during both, training and inference.
        pre_nms_topk: Number of top-K proposals to select before applying
            NMS, per FPN level. This helps in speeding up NMS.
        post_nms_topk: Number of top-K proposals to select after applying
            NMS, per FPN level. NMS is obviously going to be class-agnostic.

    Refer explanations of remaining args in the classes/functions above.
    """
    super().__init__()
    self.pred_net = RPNPredictionNetwork(
        fpn_channels, stem_channels, num_anchors=len(anchor_aspect_ratios)
    )
    # Record all input arguments:
    self.batch_size_per_image = batch_size_per_image
    self.anchor_stride_scale = anchor_stride_scale
    self.anchor_aspect_ratios = anchor_aspect_ratios
    self.anchor_iou_thresholds = anchor_iou_thresholds
    self.nms_thresh = nms_thresh
    self.pre_nms_topk = pre_nms_topk
    self.post_nms_topk = post_nms_topk

def forward(
    self,
    feats_per_fpn_level: TensorDict,
    strides_per_fpn_level: TensorDict,
    gt_boxes: Optional[torch.Tensor] = None,
):
    # Get batch size from FPN feats:
    num_images = feats_per_fpn_level["p3"].shape[0]

    ######################################################################
    # TODO: Implement the training forward pass. Follow these steps:
    #   1. Pass the FPN features per level to the RPN prediction network.
    #   2. Generate anchor boxes for all FPN levels.
    #
    # HINT: You have already implemented everything, just have to call the
    # appropriate functions.
    ######################################################################
    # Feel free to delete this line: (but keep variable names same)
    pred_obj_logits, pred_boxreg_deltas, anchors_per_fpn_level = (
        None,
        None,
        None,
    )
    # Replace "pass" statement with your code
    pass
    ######################################################################
```

```
        #                            END OF YOUR CODE                           #
        #########################################################################

        # We will fill three values in this output dict - "proposals",
        # "loss_rpn_box" (training only), "loss_rpn_obj" (training only)
        output_dict = {}

        # Get image height and width according to feature sizes and strides.
        # We need these to clamp proposals (These should be (224, 224) but we
        # avoid hard-coding them).
        img_h = feats_per_fpn_level["p3"].shape[2] * strides_per_fpn_level["p3"]
        img_w = feats_per_fpn_level["p3"].shape[3] * strides_per_fpn_level["p3"]

        # STUDENT: Implement this method before moving forward with the rest
        # of this `forward` method.
        output_dict["proposals"] = self.predict_proposals(
            anchors_per_fpn_level,
            pred_obj_logits,
            pred_boxreg_deltas,
            (img_w, img_h),
        )
        # Return here during inference - loss computation not required.
        if not self.training:
            return output_dict

        # ... otherwise continue loss computation:
        #########################################################################
        # Match the generated anchors with provided GT boxes. This
        # function is not batched so you may use a for-loop, like FCOS.
        #########################################################################
        # Combine anchor boxes from all FPN levels - we do not need any
        # distinction of boxes across different levels (for training).
        anchor_boxes = self._cat_across_fpn_levels(anchors_per_fpn_level, dim=0)

        # Get matched GT boxes (list of B tensors, each of shape `(H*W*A, 5)`
        # giving matching GT boxes to anchor boxes). Fill this list:
        matched_gt_boxes = []
        # Replace "pass" statement with your code
        pass
        #########################################################################
        #                            END OF YOUR CODE                           #
        #########################################################################

        # Combine matched boxes from all images to a `(B, HWA, 5)` tensor.
        matched_gt_boxes = torch.stack(matched_gt_boxes, dim=0)

        # Combine predictions across all FPN levels.
        pred_obj_logits = self._cat_across_fpn_levels(pred_obj_logits)
        pred_boxreg_deltas = self._cat_across_fpn_levels(pred_boxreg_deltas)

        if self.training:
            # Repeat anchor boxes `batch_size` times so there is a 1:1
            # correspondence with GT boxes.
            anchor_boxes = anchor_boxes.unsqueeze(0).repeat(num_images, 1, 1)
            anchor_boxes = anchor_boxes.contiguous().view(-1, 4)

            # Collapse `batch_size`, and `HWA` to a single dimension so we have
            # simple `(-1, 4 or 5)` tensors. This simplifies loss computation.
            matched_gt_boxes = matched_gt_boxes.view(-1, 5)
            pred_obj_logits = pred_obj_logits.view(-1)
            pred_boxreg_deltas = pred_boxreg_deltas.view(-1, 4)

            ##################################################################
            # TODO: Compute training losses. Follow three steps in order:
            #   1. Sample a few anchor boxes for training. Pass the variable
            #      `matched_gt_boxes` to `sample_rpn_training` function and
            #      use those indices to get subset of predictions and targets.
            #      RPN samples 50-50% foreground/background anchors, unless
            #      there aren't enough foreground anchors.
            #
            #   2. Compute GT targets for box regression (you have implemented
            #      the transformation function already).
            #
            #   3. Calculate objectness and box reg losses per sampled anchor.
            #      Remember to set box loss for "background" anchors to 0.
            ##################################################################
            # Feel free to delete this line: (but keep variable names same)
            loss_obj, loss_box = None, None
            # Replace "pass" statement with your code
            pass
            ##################################################################
            #                          END OF YOUR CODE                      #
            ##################################################################

            # Sum losses and average by num(foreground + background) anchors.
            # In training code, we simply add these two and call `.backward()`
            total_batch_size = self.batch_size_per_image * num_images
            output_dict["loss_rpn_obj"] = loss_obj.sum() / total_batch_size
            output_dict["loss_rpn_box"] = loss_box.sum() / total_batch_size

        return output_dict

    @torch.no_grad()  # Don't track gradients in this function.
    def predict_proposals(
        self,
        anchors_per_fpn_level: Dict[str, torch.Tensor],
        pred_obj_logits: Dict[str, torch.Tensor],
        pred_boxreg_deltas: Dict[str, torch.Tensor],
        image_size: Tuple[int, int],  # (width, height)
    ):
        """
        Predict proposals for a batch of images for the second stage. Other
        input arguments are same as those computed in `forward` method. This
        method should not be called from anywhere except from inside `forward`.

        Returns:
            torch.Tensor
                proposals: Tensor of shape `(keep_topk, 4)` giving *absolute*
                    XYXY co-ordinates of predicted proposals. These will serve
                    as anchor boxes for the second stage.
        """

        # Gather proposals from all FPN levels in this list.
        proposals_all_levels = {
            level_name: None for level_name, _ in anchors_per_fpn_level.items()
        }
        for level_name in anchors_per_fpn_level.keys():

            # Get anchor boxes and predictions from a single level.
            level_anchors = anchors_per_fpn_level[level_name]

            # shape: (batch_size, HWA), (batch_size, HWA, 4)
            level_obj_logits = pred_obj_logits[level_name]
            level_boxreg_deltas = pred_boxreg_deltas[level_name]

            # Fill proposals per image, for this FPN level, in this list.
            level_proposals_per_image = []
            for _batch_idx in range(level_obj_logits.shape[0]):
                ##############################################################
                # TODO: Perform the following steps in order:
                #   1. Transform the anchors to proposal boxes using predicted
                #      box deltas, clamp to image height and width.
                #   2. Sort all proposals by their predicted objectness, and
                #      retain `self.pre_nms_topk` proposals. This speeds up
                #      our NMS computation. HINT: `torch.topk`
                #   3. Apply NMS and retain `keep_topk_per_level` proposals
                #      per image, per level.
                #
                # NOTE: Your `nms` method may be slow for training - you may
                # use `torchvision.ops.nms` with exact same input arguments,
                # to speed up training. We will grade your `nms` implementation
                # separately; you will NOT lose points if you don't use it here.
                #
                # Note that deltas, anchor boxes, and objectness logits have
                # different shapes, you need to make some intermediate views.
                ##############################################################
                # Replace "pass" statement with your code
                pass
                ##############################################################
                #                         END OF YOUR CODE                    #
                ##############################################################

            # Collate proposals from individual images. Do not stack these
            # tensors, they may have different shapes since few images or
            # levels may have less than `post_nms_topk` proposals. We could
            # pad these tensors but there's no point - they will be used by
            # `torchvision.ops.roi_align` in second stage which operates
            # with lists, not batched tensors.
            proposals_all_levels[level_name] = level_proposals_per_image

        return proposals_all_levels

    @staticmethod
    def _cat_across_fpn_levels(
        dict_with_fpn_levels: Dict[str, torch.Tensor], dim: int = 1
    ):
        """
        Convert a dict of tensors across FPN levels {"p3", "p4", "p5"} to a
        single tensor. Values could be anything - batches of image features,
        GT targets, etc.
        """
        return torch.cat(list(dict_with_fpn_levels.values()), dim=dim)
```

```
[ ]: class FasterRCNN(nn.Module):
         """
         Faster R-CNN detector: this module combines backbone, RPN, ROI predictors.

         Unlike Faster R-CNN, we will use class-agnostic box regression and Focal
         Loss for classification. We opted for this design choice for you to re-use
         a lot of concepts that you already implemented in FCOS - choosing one loss
         over other matters less overall.
         """

         def __init__(
             self,
             backbone: nn.Module,
             rpn: nn.Module,
             stem_channels: List[int],
             num_classes: int,
             batch_size_per_image: int,
             roi_size: Tuple[int, int] = (7, 7),
         ):
             super().__init__()
             self.backbone = backbone
             self.rpn = rpn
             self.num_classes = num_classes
             self.roi_size = roi_size
             self.batch_size_per_image = batch_size_per_image

             ######################################################################
             # TODO: Create a stem of alternating 3x3 convolution layers and RELU
             # activation modules using `stem_channels` argument, exactly like
             # `FCOSPredictionNetwork` and `RPNPredictionNetwork`. use the same
             # stride, padding, and weight initialization as previous TODOs.
             #
             # HINT: This stem will be applied on RoI-aligned FPN features. You can
             # decide the number of input channels accordingly.
             ######################################################################
             # Fill this list. It is okay to use your implementation from
             # `FCOSPredictionNetwork` for this code block.
             cls_pred = []
             # Replace "pass" statement with your code
             pass

             ######################################################################
             # TODO: Add an `nn.Flatten` module to `cls_pred`, followed by a linear
             # layer to output C+1 classification logits (C classes + background).
             # Think about the input size of this linear layer based on the output
             # shape from `nn.Flatten` layer.
             ######################################################################
             # Replace "pass" statement with your code
             pass
             ######################################################################
             #                           END OF YOUR CODE                         #
             ######################################################################

             # Wrap the layers defined by student into a `nn.Sequential` module,
             # Faster R-CNN also predicts box offsets to "refine" RPN proposals, we
             # exclude it for simplicity and keep RPN proposal boxes as final boxes.
             self.cls_pred = nn.Sequential(*cls_pred)

         def forward(
             self,
             images: torch.Tensor,
             gt_boxes: Optional[torch.Tensor] = None,
             test_score_thresh: Optional[float] = None,
             test_nms_thresh: Optional[float] = None,
         ):
             """
             See documentation of `FCOS.forward` for more details.
             """

             feats_per_fpn_level = self.backbone(images)
             output_dict = self.rpn(
                 feats_per_fpn_level, self.backbone.fpn_strides, gt_boxes
             )
             proposals_per_fpn_level = output_dict["proposals"]

             # Mix GT boxes with proposals. This is necessary to stabilize training
             # since RPN proposals may be bad during first few iterations. Also, why
             # waste good supervisory signal from GT boxes, for second-stage?
             if self.training:
                 proposals_per_fpn_level = mix_gt_with_proposals(
                     proposals_per_fpn_level, gt_boxes
                 )

             # Get batch size from FPN feats:
             num_images = feats_per_fpn_level["p3"].shape[0]

             # Perform RoI-align using FPN features and proposal boxes.
             roi_feats_per_fpn_level = {
                 level_name: None for level_name in feats_per_fpn_level.keys()
             }
             # Get RPN proposals from all levels.
             for level_name in feats_per_fpn_level.keys():
                 ##################################################################
                 # TODO: Call `torchvision.ops.roi_align`. See its documentation to
                 # properly format input arguments. Use `aligned=True`
                 ##################################################################
                 level_feats = feats_per_fpn_level[level_name]
                 level_props = output_dict["proposals"][level_name]
                 level_stride = self.backbone.fpn_strides[level_name]

                 # Replace "pass" statement with your code
                 pass
                 ##################################################################
                 #                         END OF YOUR CODE                       #
                 ##################################################################

                 roi_feats_per_fpn_level[level_name] = roi_feats

             # Combine ROI feats across FPN levels, do the same with proposals.
             # shape: (batch_size * total_proposals, fpn_channels, roi_h, roi_w)
             roi_feats = self._cat_across_fpn_levels(roi_feats_per_fpn_level, dim=0)

             # Obtain classification logits for all ROI features.
             # shape: (batch_size * total_proposals, num_classes)
             pred_cls_logits = self.cls_pred(roi_feats)

             if not self.training:
                 # During inference, just go to this method and skip rest of the
                 # forward pass. Batch size must be 1!
                 # fmt: off
                 return self.inference(
                     images,
                     proposals_per_fpn_level,
                     pred_cls_logits,
                     test_score_thresh=test_score_thresh,
                     test_nms_thresh=test_nms_thresh,
                 )
                 # fmt: on

             ######################################################################
             # Match the RPN proposals with provided GT boxes and append to
             # `matched_gt_boxes`. Use `rcnn_match_anchors_to_gt` with IoU threshold
             # such that IoU > 0.5 is foreground, otherwise background.
             # There are no neutral proposals in second-stage.
             ######################################################################
             matched_gt_boxes = []
             for _idx in range(len(gt_boxes)):
                 # Get proposals per image from this dictionary of list of tensors.
                 proposals_per_fpn_level_per_image = {
                     level_name: prop[_idx]
                     for level_name, prop in output_dict["proposals"].items()
                 }
                 proposals_per_image = self._cat_across_fpn_levels(
                     proposals_per_fpn_level_per_image, dim=0
                 )
                 gt_boxes_per_image = gt_boxes[_idx]
                 # Replace "pass" statement with your code
                 pass
             ######################################################################
             #                             END OF YOUR CODE                       #
             ######################################################################

             # Combine predictions and GT from across all FPN levels.
             matched_gt_boxes = torch.cat(matched_gt_boxes, dim=0)

             ######################################################################
             # TODO: Train the classifier head. Perform these steps in order:
             #   1. Sample a few RPN proposals, like you sampled 50-50% anchor boxes
             #      to train RPN objectness classifier. However this time, sample
             #      such that ~25% RPN proposals are foreground, and the rest are
             #      background. Faster R-CNN performed such weighted sampling to
             #      deal with class imbalance, before Focal Loss was published.
             #
             #   2. Use these indices to get GT class labels from `matched_gt_boxes`
             #      and obtain the corresponding logits predicted by classifier.
             #
             #   3. Compute cross entropy loss - use `F.cross_entropy`, see its API
             #      documentation on PyTorch website. Since background ID = -1, you
             #      may shift class labels by +1 such that background ID = 0 and
             #      other VC classes have IDs (1-20). Make sure to reverse shift
             #      this during inference, so that model predicts VOC IDs (0-19).
             ######################################################################
             # Feel free to delete this line: (but keep variable names same)
             loss_cls = None
             # Replace "pass" statement with your code
             pass
             ######################################################################
             #                             END OF YOUR CODE                       #
             ######################################################################
             return {
                 "loss_rpn_obj": output_dict["loss_rpn_obj"],
                 "loss_rpn_box": output_dict["loss_rpn_box"],
                 "loss_cls": loss_cls,
             }

         @staticmethod
         def _cat_across_fpn_levels(
             dict_with_fpn_levels: Dict[str, torch.Tensor], dim: int = 1
         ):
             """
             Convert a dict of tensors across FPN levels {"p3", "p4", "p5"} to a
             single tensor. Values could be anything - batches of image features,
             GT targets, etc.
             """
             return torch.cat(list(dict_with_fpn_levels.values()), dim=dim)

         def inference(
             self,
             images: torch.Tensor,
             proposals: torch.Tensor,
             pred_cls_logits: torch.Tensor,
             test_score_thresh: float,
             test_nms_thresh: float,
         ):
             """
             Run inference on a single input image (batch size = 1). Other input
             arguments are same as those computed in `forward` method. This method
             should not be called from anywhere except from inside `forward`.

             Returns:
                 Three tensors:
                     - pred_boxes: Tensor of shape `(N, 4)` giving *absolute* XYXY
                       co-ordinates of predicted boxes.

                     - pred_classes: Tensor of shape `(N, )` giving predicted class
                       labels for these boxes (one of `num_classes` labels). Make
                       sure there are no background predictions (-1).

                     - pred_scores: Tensor of shape `(N, )` giving confidence scores
                       for predictions.
             """

             # The second stage inference in Faster R-CNN is quite straightforward:
             # combine proposals from all FPN levels and perform a *class-specific
             # NMS*. There would have been more steps here if we further refined
             # RPN proposals by predicting box regression deltas.

             # Use `[0]` to remove the batch dimension.
             proposals = {level_name: prop[0] for level_name, prop in proposals.
     ↪items()}
             pred_boxes = self._cat_across_fpn_levels(proposals, dim=0)

             ######################################################################
             # Faster R-CNN inference, perform the following steps in order:
             #   1. Get the most confident predicted class and score for every box.
             #      Note that the "score" of any class (including background) is its
             #      probability after applying C+1 softmax.
             #
             #   2. Only retain prediction that have a confidence score higher than
             #      provided threshold in arguments.
             #
             # NOTE: `pred_classes` may contain background as ID = 0 (based on how
             # the classifier was supervised in `forward`). Remember to shift the
             # predicted IDs such that model outputs ID (0-19) for 20 VOC classes.
             ######################################################################
             pred_scores, pred_classes = None, None
             # Replace "pass" statement with your code
             pass
             ######################################################################
             #                             END OF YOUR CODE                       #
             ######################################################################

             # STUDENTS: This line depends on your implementation of NMS.
             keep = class_spec_nms(
                 pred_boxes, pred_scores, pred_classes, iou_threshold=test_nms_thresh
             )
             pred_boxes = pred_boxes[keep]
             pred_classes = pred_classes[keep]
             pred_scores = pred_scores[keep]
             return pred_boxes, pred_classes, pred_scores
```