one_stage_detector

September 7, 2024

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

[ ]:
```
import torch
from a4_helper import *
from common import DetectorBackboneWithFPN, class_spec_nms,␣
↪get_fpn_location_coords
from torch import nn
from torch.nn import functional as F
from torch.utils.data._utils.collate import default_collate
from torchvision.ops import sigmoid_focal_loss
```

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

[ ]:
```
def hello_one_stage_detector():
    print("Hello from one_stage_detector.py!")
```

[ ]:
```
class FCOSPredictionNetwork(nn.Module):
    """
    FCOS prediction network that accepts FPN feature maps from different levels
    and makes three predictions at every location: bounding boxes, class ID and
    centerness. This module contains a "stem" of convolution layers, along with
    one final layer per prediction. For a visual depiction, see Figure 2 (right
    side) in FCOS paper: https://arxiv.org/abs/1904.01355

    We will use feature maps from FPN levels (P3, P4, P5) and exclude (P6, P7).
    """

    def __init__(
        self, num_classes: int, in_channels: int, stem_channels: List[int]
    ):
        """
        Args:
            num_classes: Number of object classes for classification.
            in_channels: Number of channels in input feature maps. This value
                is same as the output channels of FPN, since the head directly
                operates on them.
```

```
        stem_channels: List of integers giving the number of output channels
            in each convolution layer of stem layers.
        """
        super().__init__()

        ######################################################################
        # TODO: Create a stem of alternating 3x3 convolution layers and RELU
        # activation modules. Note there are two separate stems for class and
        # box stem. The prediction layers for box regression and centerness
        # operate on the output of `stem_box`.
        # See FCOS figure again; both stems are identical.
        #
        # 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 these.
        stem_cls = []
        stem_box = []
        # Replace "pass" statement with your code
        pass

        # Wrap the layers defined by student into a `nn.Sequential` module:
        self.stem_cls = nn.Sequential(*stem_cls)
        self.stem_box = nn.Sequential(*stem_box)

        ######################################################################
        # TODO: Create THREE 3x3 conv layers for individually predicting three
        # things at every location of feature map:
        #     1. object class logits (`num_classes` outputs)
        #     2. box regression deltas (4 outputs: LTRB deltas from locations)
        #     3. centerness logits (1 output)
        #
        # Class probability and actual centerness are obtained by applying
        # sigmoid activation to these logits. However, DO NOT initialize those
        # modules here. This module should always output logits; PyTorch loss
        # functions have numerically stable implementations with logits. During
        # inference, logits are converted to probabilities by applying sigmoid,
        # BUT OUTSIDE this module.
        #
        ######################################################################

        # Replace these lines with your code, keep variable names unchanged.
        self.pred_cls = None  # Class prediction conv
```

```
        self.pred_box = None  # Box regression conv
        self.pred_ctr = None  # Centerness conv

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

        # OVERRIDE: Use a negative bias in `pred_cls` to improve training
        # stability. Without this, the training will most likely diverge.
        # STUDENTS: You do not need to get into details of why this is needed.
        torch.nn.init.constant_(self.pred_cls.bias, -math.log(99))

    def forward(self, feats_per_fpn_level: TensorDict) -> List[TensorDict]:
        """
        Accept FPN feature maps and predict the desired outputs at every␣
↪location
        (as described above). Format them such that channels are placed at the
        last dimension, and (H, W) are flattened (having channels at last is
        convenient for computing loss as well as performing inference).

        Args:
            feats_per_fpn_level: Features from FPN, keys {"p3", "p4", "p5"}.␣
↪Each
                tensor will have shape `(batch_size, fpn_channels, H, W)`. For␣
↪an
                input (224, 224) image, H = W are (28, 14, 7) for (p3, p4, p5).

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

        ##########################################################################
        # TODO: Iterate over every FPN feature map and obtain predictions using
        # the layers defined above. Remember that prediction layers of box
        # regression and centerness will operate on output of `stem_box`,
        # and classification layer operates separately on `stem_cls`.
        #
        # CAUTION: The original FCOS model uses shared stem for centerness and
        # classification. Recent follow-up papers commonly place centerness and
        # box regression predictors with a shared stem, which we follow here.
        #
        # DO NOT apply sigmoid to classification and centerness logits.
```

```
        ##########################################################################
        # Fill these with keys: {"p3", "p4", "p5"}, same as input dictionary.
        class_logits = {}
        boxreg_deltas = {}
        centerness_logits = {}

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

        return [class_logits, boxreg_deltas, centerness_logits]
```

[ ]:
```
@torch.no_grad()
def fcos_match_locations_to_gt(
    locations_per_fpn_level: TensorDict,
    strides_per_fpn_level: Dict[str, int],
    gt_boxes: torch.Tensor,
) -> TensorDict:
    """
    Match centers of the locations of FPN feature with a set of GT bounding
    boxes of the input image. Since our model makes predictions at every FPN
    feature map location, we must supervise it with an appropriate GT box.
    There are multiple GT boxes in image, so FCOS has a set of heuristics to
    assign centers with GT, which we implement here.

    NOTE: This function is NOT BATCHED. Call separately for GT box batches.

    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 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.
        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.

    Returns:
        Dict[str, torch.Tensor]
            Dictionary with same keys as `shape_per_fpn_level` and values as
            tensors of shape `(N, 5)` GT boxes, one for each center. They are
            one of M input boxes, or a dummy box called "background" that is
```

```
            `(-1, -1, -1, -1, -1)`. Background indicates that the center does
            not belong to any object.
    """

    matched_gt_boxes = {
        level_name: None for level_name in locations_per_fpn_level.keys()
    }

    # Do this matching individually per FPN level.
    for level_name, centers in locations_per_fpn_level.items():

        # Get stride for this FPN level.
        stride = strides_per_fpn_level[level_name]

        x, y = centers.unsqueeze(dim=2).unbind(dim=1)
        x0, y0, x1, y1 = gt_boxes[:, :4].unsqueeze(dim=0).unbind(dim=2)
        pairwise_dist = torch.stack([x - x0, y - y0, x1 - x, y1 - y], dim=2)

        # Pairwise distance between every feature center and GT box edges:
        # shape: (num_gt_boxes, num_centers_this_level, 4)
        pairwise_dist = pairwise_dist.permute(1, 0, 2)

        # The original FCOS anchor matching rule: anchor point must be inside␣
↪GT.
        match_matrix = pairwise_dist.min(dim=2).values > 0

        # Multilevel anchor matching in FCOS: each anchor is only responsible
        # for certain scale range.
        # Decide upper and lower bounds of limiting targets.
        pairwise_dist = pairwise_dist.max(dim=2).values

        lower_bound = stride * 4 if level_name != "p3" else 0
        upper_bound = stride * 8 if level_name != "p5" else float("inf")
        match_matrix &= (pairwise_dist > lower_bound) & (
            pairwise_dist < upper_bound
        )

        # Match the GT box with minimum area, if there are multiple GT matches.
        gt_areas = (gt_boxes[:, 2] - gt_boxes[:, 0]) * (
            gt_boxes[:, 3] - gt_boxes[:, 1]
        )

        # Get matches and their labels using match quality matrix.
        match_matrix = match_matrix.to(torch.float32)
        match_matrix *= 1e8 - gt_areas[:, None]

        # Find matched ground-truth instance per anchor (un-matched = -1).
```

```
        match_quality, matched_idxs = match_matrix.max(dim=0)
        matched_idxs[match_quality < 1e-5] = -1

        # Anchors with label 0 are treated as background.
        matched_boxes_this_level = gt_boxes[matched_idxs.clip(min=0)]
        matched_boxes_this_level[matched_idxs < 0, :] = -1

        matched_gt_boxes[level_name] = matched_boxes_this_level

    return matched_gt_boxes
```

[ ]:
```
def fcos_get_deltas_from_locations(
    locations: torch.Tensor, gt_boxes: torch.Tensor, stride: int
) -> torch.Tensor:
    """
    Compute distances from feature locations to GT box edges. These distances
    are called "deltas" - `(left, top, right, bottom)` or simply `LTRB`. The
    feature locations and GT boxes are given in absolute image co-ordinates.

    These deltas are used as targets for training FCOS to perform box regression
    and centerness regression. They must be "normalized" by the stride of FPN
    feature map (from which feature locations were computed, see the function
    `get_fpn_location_coords`). If GT boxes are "background", then deltas must
    be `(-1, -1, -1, -1)`.

    NOTE: This transformation function should not require GT class label. Your
    implementation must work for GT boxes being `(N, 4)` or `(N, 5)` tensors -
    without or with class labels respectively. You may assume that all the
    background boxes will be `(-1, -1, -1, -1)` or `(-1, -1, -1, -1, -1)`.

    Args:
        locations: Tensor of shape `(N, 2)` giving `(xc, yc)` feature locations.
        gt_boxes: Tensor of shape `(N, 4 or 5)` giving GT boxes.
        stride: Stride of the FPN feature map.

    Returns:
        torch.Tensor
            Tensor of shape `(N, 4)` giving deltas from feature locations, that
            are normalized by feature stride.
    """
    ##########################################################################
    # TODO: Implement the logic to get deltas from feature locations.        #
    ##########################################################################
    # Set this to Tensor of shape (N, 4) giving deltas (left, top, right,␣
↪bottom)
    # from the locations to GT box edges, normalized by FPN stride.
    deltas = None
```

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

    return deltas
```

[ ]:
```
def fcos_apply_deltas_to_locations(
    deltas: torch.Tensor, locations: torch.Tensor, stride: int
) -> torch.Tensor:
    """
    Implement the inverse of `fcos_get_deltas_from_locations` here:

    Given edge deltas (left, top, right, bottom) and feature locations of FPN, get
    the resulting bounding box co-ordinates by applying deltas on locations. This
    method is used for inference in FCOS: deltas are outputs from model, and
    applying them to anchors will give us final box predictions.

    Recall in above method, we were required to normalize the deltas by feature
    stride. Similarly, we have to un-normalize the input deltas with feature
    stride before applying them to locations, because the given input locations are
    already absolute co-ordinates in image dimensions.

    Args:
        deltas: Tensor of shape `(N, 4)` giving edge deltas to apply to locations.
        locations: Locations to apply deltas on. shape: `(N, 2)`
        stride: Stride of the FPN feature map.

    Returns:
        torch.Tensor
            Same shape as deltas and locations, giving co-ordinates of the
            resulting boxes `(x1, y1, x2, y2)`, absolute in image dimensions.
    """
    ##########################################################################
    # TODO: Implement the transformation logic to get boxes.                 #
    #                                                                        #
    # NOTE: The model predicted deltas MAY BE negative, which is not valid   #
    # for our use-case because the feature center must lie INSIDE the final  #
    # box. Make sure to clip them to zero.                                   #
    ##########################################################################
    # Replace "pass" statement with your code
```

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

    return output_boxes
```

[ ]:
```
def fcos_make_centerness_targets(deltas: torch.Tensor):
    """
    Given LTRB deltas of GT boxes, compute GT targets for supervising the
    centerness regression predictor. See `fcos_get_deltas_from_locations` on
    how deltas are computed. If GT boxes are "background" => deltas are
    `(-1, -1, -1, -1)`, then centerness should be `-1`.

    For reference, centerness equation is available in FCOS paper
    https://arxiv.org/abs/1904.01355 (Equation 3).

    Args:
        deltas: Tensor of shape `(N, 4)` giving LTRB deltas for GT boxes.

    Returns:
        torch.Tensor
            Tensor of shape `(N, )` giving centerness regression targets.
    """
    ##########################################################################
    # TODO: Implement the centerness calculation logic.                      #
    ##########################################################################
    centerness = None
    # Replace "pass" statement with your code
    pass
    ##########################################################################
    #                            END OF YOUR CODE                            #
    ##########################################################################

    return centerness
```

[ ]:
```
class FCOS(nn.Module):
    """
    FCOS: Fully-Convolutional One-Stage Detector

    This class puts together everything you implemented so far. It contains a
    backbone with FPN, and prediction layers (head). It computes loss during
    training and predicts boxes during inference.
    """

    def __init__(
        self, num_classes: int, fpn_channels: int, stem_channels: List[int]
```

```
    ):
        super().__init__()
        self.num_classes = num_classes

        ######################################################################
        # TODO: Initialize backbone and prediction network using arguments.  #
        ######################################################################
        # Feel free to delete these two lines: (but keep variable names same)
        self.backbone = None
        self.pred_net = None
        # Replace "pass" statement with your code
        pass
        ######################################################################
        #                           END OF YOUR CODE                         #
        ######################################################################

        # Averaging factor for training loss; EMA of foreground locations.
        # STUDENTS: See its use in `forward` when you implement losses.
        self._normalizer = 150  # per image

    def forward(
        self,
        images: torch.Tensor,
        gt_boxes: Optional[torch.Tensor] = None,
        test_score_thresh: Optional[float] = None,
        test_nms_thresh: Optional[float] = None,
    ):
        """
        Args:
            images: Batch of images, tensors of shape `(B, C, H, W)`.
            gt_boxes: Batch of training boxes, tensors of shape `(B, N, 5)`.
                `gt_boxes[i, j] = (x1, y1, x2, y2, C)` gives information about
                the `j`th object in `images[i]`. The position of the top-left
                corner of the box is `(x1, y1)` and the position of bottom-right
                corner of the box is `(x2, x2)`. These coordinates are
                real-valued in `[H, W]`. `C` is an integer giving the category
                label for this bounding box. Not provided during inference.
            test_score_thresh: During inference, discard predictions with a
                confidence score less than this value. Ignored during training.
            test_nms_thresh: IoU threshold for NMS during inference. Ignored
                during training.

        Returns:
            Losses during training and predictions during inference.
        """

        ######################################################################
```

```
        # TODO: Process the image through backbone, FPN, and prediction head #
        # to obtain model predictions at every FPN location.                  #
        # Get dictionaries of keys {"p3", "p4", "p5"} giving predicted class #
        # logits, deltas, and centerness.                                     #
        ######################################################################
        # Feel free to delete this line: (but keep variable names same)
        pred_cls_logits, pred_boxreg_deltas, pred_ctr_logits = None, None, None
        # Replace "pass" statement with your code
        pass

        ######################################################################
        # TODO: Get absolute co-ordinates `(xc, yc)` for every location in
        # FPN levels.
        #
        # HINT: You have already implemented everything, just have to
        # call the functions properly.
        ######################################################################
        # Feel free to delete this line: (but keep variable names same)
        locations_per_fpn_level = None
        # Replace "pass" statement with your code
        pass
        ######################################################################
        #                            END OF YOUR CODE                        #
        ######################################################################

        if not self.training:
            # During inference, just go to this method and skip rest of the
            # forward pass.
            # fmt: off
            return self.inference(
                images, locations_per_fpn_level,
                pred_cls_logits, pred_boxreg_deltas, pred_ctr_logits,
                test_score_thresh=test_score_thresh,
                test_nms_thresh=test_nms_thresh,
            )
            # fmt: on

        ######################################################################
        # TODO: Assign ground-truth boxes to feature locations. We have this
        # implemented in a `fcos_match_locations_to_gt`. This operation is NOT
        # batched so call it separately per GT boxes in batch.
        ######################################################################
        # List of dictionaries with keys {"p3", "p4", "p5"} giving matched
        # boxes for locations per FPN level, per image. Fill this list:
        matched_gt_boxes = []
        # Replace "pass" statement with your code
        pass
```

```
        # Calculate GT deltas for these matched boxes. Similar structure
        # as `matched_gt_boxes` above. Fill this list:
        matched_gt_deltas = []
        # Replace "pass" statement with your code
        pass
        ##########################################################################
        #                            END OF YOUR CODE                            #
        ##########################################################################

        # Collate lists of dictionaries, to dictionaries of batched tensors.
        # These are dictionaries with keys {"p3", "p4", "p5"} and values as
        # tensors of shape (batch_size, locations_per_fpn_level, 5 or 4)
        matched_gt_boxes = default_collate(matched_gt_boxes)
        matched_gt_deltas = default_collate(matched_gt_deltas)

        # Combine predictions and GT from across all FPN levels.
        # shape: (batch_size, num_locations_across_fpn_levels, ...)
        matched_gt_boxes = self._cat_across_fpn_levels(matched_gt_boxes)
        matched_gt_deltas = self._cat_across_fpn_levels(matched_gt_deltas)
        pred_cls_logits = self._cat_across_fpn_levels(pred_cls_logits)
        pred_boxreg_deltas = self._cat_across_fpn_levels(pred_boxreg_deltas)
        pred_ctr_logits = self._cat_across_fpn_levels(pred_ctr_logits)

        # Perform EMA update of normalizer by number of positive locations.
        num_pos_locations = (matched_gt_boxes[:, :, 4] != -1).sum()
        pos_loc_per_image = num_pos_locations.item() / images.shape[0]
        self._normalizer = 0.9 * self._normalizer + 0.1 * pos_loc_per_image

        ##########################################################################
        # TODO: Calculate losses per location for classification, box reg and
        # centerness. Remember to set box/centerness losses for "background"
        # positions to zero.
        ##########################################################################
        # Feel free to delete this line: (but keep variable names same)
        loss_cls, loss_box, loss_ctr = None, None, None

        # Replace "pass" statement with your code
        pass
        ##########################################################################
        #                            END OF YOUR CODE                            #
        ##########################################################################
        # Sum all locations and average by the EMA of foreground locations.
        # In training code, we simply add these three and call `.backward()`
        return {
            "loss_cls": loss_cls.sum() / (self._normalizer * images.shape[0]),
            "loss_box": loss_box.sum() / (self._normalizer * images.shape[0]),
```

```
            "loss_ctr": loss_ctr.sum() / (self._normalizer * images.shape[0]),
        }

    @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,
        locations_per_fpn_level: Dict[str, torch.Tensor],
        pred_cls_logits: Dict[str, torch.Tensor],
        pred_boxreg_deltas: Dict[str, torch.Tensor],
        pred_ctr_logits: Dict[str, torch.Tensor],
        test_score_thresh: float = 0.3,
        test_nms_thresh: float = 0.5,
    ):
        """
        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: these values are `sqrt(class_prob * ctrness)`
                  where class_prob and ctrness are obtained by applying sigmoid
                  to corresponding logits.
        """

        # Gather scores and boxes from all FPN levels in this list. Once
        # gathered, we will perform NMS to filter highly overlapping␣
↪predictions.
```

```
        pred_boxes_all_levels = []
        pred_classes_all_levels = []
        pred_scores_all_levels = []

        for level_name in locations_per_fpn_level.keys():

            # Get locations and predictions from a single level.
            # We index predictions by `[0]` to remove batch dimension.
            level_locations = locations_per_fpn_level[level_name]
            level_cls_logits = pred_cls_logits[level_name][0]
            level_deltas = pred_boxreg_deltas[level_name][0]
            level_ctr_logits = pred_ctr_logits[level_name][0]

            ##########################################################################
            # TODO: FCOS uses the geometric mean of class probability and
            # centerness as the final confidence score. This helps in getting
            # rid of excessive amount of boxes far away from object centers.
            # Compute this value here (recall sigmoid(logits) = probabilities)
            #
            # Then perform the following steps in order:
            #   1. Get the most confidently predicted class and its score for
            #      every box. Use level_pred_scores: (N, num_classes) => (N, )
            #   2. Only retain prediction that have a confidence score higher
            #      than provided threshold in arguments.
            #   3. Obtain predicted boxes using predicted deltas and locations
            #   4. Clip XYXY box-cordinates that go beyond thr height and
            #      and width of input image.
            ##########################################################################
            # Feel free to delete this line: (but keep variable names same)
            level_pred_boxes, level_pred_classes, level_pred_scores = (
                None,
                None,
                None,  # Need tensors of shape: (N, 4) (N, ) (N, )
            )

            # Compute geometric mean of class logits and centerness:
            level_pred_scores = torch.sqrt(
                level_cls_logits.sigmoid_() * level_ctr_logits.sigmoid_()
            )
            # Step 1:
            # Replace "pass" statement with your code
            pass

            # Step 2:
            # Replace "pass" statement with your code
            pass
```

```
            # Step 3:
            # Replace "pass" statement with your code
            pass

            # Step 4: Use `images` to get (height, width) for clipping.
            # Replace "pass" statement with your code
            pass
            ##########################################################################
            #                            END OF YOUR CODE                            #
            ##########################################################################

            pred_boxes_all_levels.append(level_pred_boxes)
            pred_classes_all_levels.append(level_pred_classes)
            pred_scores_all_levels.append(level_pred_scores)

        ##############################################################################
        # Combine predictions from all levels and perform NMS.
        pred_boxes_all_levels = torch.cat(pred_boxes_all_levels)
        pred_classes_all_levels = torch.cat(pred_classes_all_levels)
        pred_scores_all_levels = torch.cat(pred_scores_all_levels)

        # STUDENTS: This function depends on your implementation of NMS.
        keep = class_spec_nms(
            pred_boxes_all_levels,
            pred_scores_all_levels,
            pred_classes_all_levels,
            iou_threshold=test_nms_thresh,
        )
        pred_boxes_all_levels = pred_boxes_all_levels[keep]
        pred_classes_all_levels = pred_classes_all_levels[keep]
        pred_scores_all_levels = pred_scores_all_levels[keep]
        return (
            pred_boxes_all_levels,
            pred_classes_all_levels,
            pred_scores_all_levels,
        )
```