st125457_Ulugbek_a2_object_detection_yolov4

February 4, 2025

1 RTML | A2 Object Detection

1.0.1 Ulugbek Shernazarov - st125457

1.0.2 Object Detection

If you could go back in time to the 1990s, there were no cameras that could find faces in a photograph, and no researcher had a way to count dogs in a video in real time. Everyone had to count the dogs manually. Times were very tough.

The Holy Grail of computer vision research at the time was real time face detection. If we could find faces in images fast enough, we could build systems that interact more naturally with human beings. But nobody had a solution.

Things changed when Viola and Jones introduced the first real time face detector, the Haar-like cascade, at the end of the 1990s. This technique swept a detection window over the input image at multiple sizes, and subjected each local patch to a cascade of simple rough classifiers. Each patch that made it to the end of the cascade of classifiers was treated as a positive detection. After a set of candidate patches were identified, there would be a cleanup stage when neighboring detections are clustered into isolated detections.

This method and one cousin, the HOG detector, which was slower but a little more accurate, dominated during the 2000s and on into the 2010s. These methods worked well enough when trained carefully on the specific environment they were used in, but usually couldn't be transfer to a new environment.

With the introduction of AlexNet and the amazing advances in image classification, we could follow the direction of R-CNN, to use a region proposal algorithm followed by a deep learning classifier to do object detection VERY slowly but much more accurately than the old real time methods.

1.0.3 Task 1: Inference

In the lab, we saw how the Darknet configuration file for YOLOv3 could be read in Python and mapped to PyTorch modules.

For your independent work do the same thing for YOLOv4. Download the `yolov4.cfg` file from the YOLOv4 GitHub repository and modify your `MyDarknet` class and utility code (`darknet.py`, `util.py`) as necessary to map the structures to PyTorch.

The changes you'll have to make:

1. Implement the mish activation function - Done
2. Add an option for a maxpool layer in the `create_modules` function and in your model's `forward()` method. - Done
3. Enable a `[route]` module to concatenate more than two previous layers - Done
4. Load the pre-trained weights provided by the authors - Done
5. Scale inputs to 608$×$608 and make sure you're passing input channels in RGB order, not OpenCV's BGR order. - Done

[ ]:
```
# mish.py
# 1. Implementation of the mish activation function
import torch
import torch.nn as nn
import torch.nn.functional as F

class Mish(nn.Module):
    """
    Mish activation function implementation:
        mish = x * tanh(softplus(x)) = x * tanh(ln(1 + exp(x)))
    """
    def __init__(self):
        super(Mish, self).__init__()

    def forward(self, x):
        return x * torch.tanh(F.softplus(x))
```

[ ]:
```
# darknet.py
# 2. Add an option for a maxpool layer in the `create_modules` function and in␣
↪your model's `forward()` method.
# 3. Enable a `[route]` module to concatenate more than two previous layers
from __future__ import division

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import numpy as np
from utils.mish import Mish
from utils.util import *



def get_test_input():
    img = cv2.imread("dog-cycle-car.png")
    img = cv2.resize(img, (608,608))          #Resize to the input dimension
    img_ =  img[:,:,::-1].transpose((2,0,1))  # BGR -> RGB | H X W C -> C X H X␣
↪W
```

```python
    img_ = img_[np.newaxis,:,:,:]/255.0       #Add a channel at 0 (for batch) | Normalise
    img_ = torch.from_numpy(img_).float()     #Convert to float
    img_ = Variable(img_)                     # Convert to Variable
    return img_

def parse_cfg(cfgfile):
    """
    Takes a configuration file

    Returns a list of blocks. Each blocks describes a block in the neural
    network to be built. Block is represented as a dictionary in the list

    """

    file = open(cfgfile, 'r')
    lines = file.read().split('\n')                        # store the lines in a list
    lines = [x for x in lines if len(x) > 0]               # get read of the empty lines
    lines = [x for x in lines if x[0] != '#']              # get rid of comments
    lines = [x.rstrip().lstrip() for x in lines]           # get rid of fringe whitespaces

    block = {}
    blocks = []

    for line in lines:
        if line[0] == "[":               # This marks the start of a new block
            if len(block) != 0:          # If block is not empty, implies it is storing values of previous block.
                blocks.append(block)     # add it the blocks list
                block = {}               # re-init the block
            block["type"] = line[1:-1].rstrip()
        else:
            key,value = line.split("=")
            block[key.rstrip()] = value.lstrip()
    blocks.append(block)

    return blocks

def get_anchors(blocks):
    """Extract anchors from YOLO layers in config blocks"""
    anchors = []
    for block in blocks:
        if block["type"] == "yolo":
            if "anchors" in block:
```

```
                # Parse anchor string into list of tuples
                anchor_pairs = [float(x) for x in block["anchors"].split(",")]
                block_anchors = [(anchor_pairs[i], anchor_pairs[i+1]) for i in␣
↪range(0, len(anchor_pairs), 2)]
                anchors.extend(block_anchors)
    return anchors

def prep_image(img, inp_dim):
    """
    Prepare image for inputting to the neural network. Returns a tensor.
    """
    # pylint: disable=no-member
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = letterbox_image(img, (inp_dim, inp_dim))
    return torch.from_numpy(img.transpose(2, 0, 1)).float().div(255.0).
↪unsqueeze(0)


class EmptyLayer(nn.Module):
    def __init__(self):
        super(EmptyLayer, self).__init__()


class DetectionLayer(nn.Module):
    def __init__(self, anchors):
        super(DetectionLayer, self).__init__()
        self.anchors = anchors



def create_modules(blocks):
    net_info = blocks[0]     #Captures the information about the input and␣
↪pre-processing
    module_list = nn.ModuleList()
    prev_filters = 3
    output_filters = []

    for index, x in enumerate(blocks[1:]):
        module = nn.Sequential()

        #check the type of block
        #create a new module for the block
        #append to module_list

        #If it's a convolutional layer
        if (x["type"] == "convolutional"):
            #Get the info about the layer
```

```python
            activation = x["activation"]
            try:
                batch_normalize = int(x["batch_normalize"])
                bias = False
            except:
                batch_normalize = 0
                bias = True

            filters= int(x["filters"])
            padding = int(x["pad"])
            kernel_size = int(x["size"])
            stride = int(x["stride"])

            if padding:
                pad = (kernel_size - 1) // 2
            else:
                pad = 0

            #Add the convolutional layer
            conv = nn.Conv2d(prev_filters, filters, kernel_size, stride, pad, bias = bias)
            module.add_module("conv_{0}".format(index), conv)

            #Add the Batch Norm Layer
            if batch_normalize:
                bn = nn.BatchNorm2d(filters)
                module.add_module("batch_norm_{0}".format(index), bn)

            #Check the activation.
            #It is either Linear or a Leaky ReLU for YOLO
            if activation == "leaky":
                activn = nn.LeakyReLU(0.1, inplace = True)
                module.add_module("leaky_{0}".format(index), activn)

            elif activation == "mish":
                activn = Mish()
                module.add_module("mish_{0}".format(index), activn)


            #If it's an upsampling layer
            #We use Bilinear2dUpsampling
        elif (x["type"] == "upsample"):
            stride = int(x["stride"])
            upsample = nn.Upsample(scale_factor = 2, mode = "nearest")
            module.add_module("upsample_{}".format(index), upsample)

        #If it is a route layer
```

```
        elif (x["type"] == "route"):
            x["layers"] = x["layers"].split(',')
            filters = 0

            for i in range(len(x["layers"])):
                pointer = int(x["layers"][i])
                if  pointer > 0:
                    filters += output_filters[pointer]
                else:
                    filters += output_filters[index + pointer]

            route = EmptyLayer()
            module.add_module("route_{0}".format(index), route)

        #shortcut corresponds to skip connection
        elif x["type"] == "shortcut":
            shortcut = EmptyLayer()
            module.add_module("shortcut_{}".format(index), shortcut)

        #Yolo is the detection layer
        elif x["type"] == "yolo":
            mask = x["mask"].split(",")
            mask = [int(x) for x in mask]

            anchors = x["anchors"].split(",")
            anchors = [int(a) for a in anchors]
            anchors = [(anchors[i], anchors[i+1]) for i in range(0, len(anchors),2)]
            anchors = [anchors[i] for i in mask]

            detection = DetectionLayer(anchors)
            module.add_module("Detection_{}".format(index), detection)

        # Max pooling layer
        elif x["type"] == "maxpool":
            stride = int(x["stride"])
            size = int(x["size"])
            max_pool = nn.MaxPool2d(size, stride, padding=size // 2)
            module.add_module("maxpool_{}".format(index), max_pool)

        module_list.append(module)
        prev_filters = filters
        output_filters.append(filters)

    return (net_info, module_list)

class Darknet(nn.Module):
```

```python
    def __init__(self, cfgfile):
        super(Darknet, self).__init__()
        self.blocks = parse_cfg(cfgfile)
        self.net_info, self.module_list = create_modules(self.blocks)
        self.anchors = get_anchors(self.blocks)  # Parse and store anchors

    def forward(self, x, CUDA):
        modules = self.blocks[1:]
        outputs = {}   #We cache the outputs for the route layer

        write = 0
        for i, module in enumerate(modules):
            module_type = (module["type"])


            if module_type in ["convolutional", "upsample", "maxpool"]:
                x = self.module_list[i](x)

            elif module_type == "route":
                layers = module["layers"]
                layers = [int(a) for a in layers]
                maps = []
                for l in range(0, len(layers)):
                    if layers[l] > 0:
                        layers[l] = layers[l] - i
                    maps.append(outputs[i + layers[l]])
                x = torch.cat((maps), 1)


            elif  module_type == "shortcut":
                from_ = int(module["from"])
                x = outputs[i-1] + outputs[i+from_]

            elif module_type == 'yolo':
                anchors = self.module_list[i][0].anchors
                #Get the input dimensions
                inp_dim = int (self.net_info["height"])

                #Get the number of classes
                num_classes = int (module["classes"])

                #Transform
                x = x.data
                x = predict_transform(x, inp_dim, anchors, num_classes, CUDA)
                if not write:              #if no collector has been intialised.
↪
                    detections = x
```

```
                write = 1

            else:
                detections = torch.cat((detections, x), 1)

        outputs[i] = x

    return detections


def load_weights(self, weightfile):
    #Open the weights file
    fp = open(weightfile, "rb")

    #The first 5 values are header information
    # 1. Major version number
    # 2. Minor Version Number
    # 3. Subversion number
    # 4,5. Images seen by the network (during training)
    header = np.fromfile(fp, dtype = np.int32, count = 5)
    self.header = torch.from_numpy(header)
    self.seen = self.header[3]

    weights = np.fromfile(fp, dtype = np.float32)

    ptr = 0
    for i in range(len(self.module_list)):
        module_type = self.blocks[i + 1]["type"]

        #If module_type is convolutional load weights
        #Otherwise ignore.

        if module_type == "convolutional":
            model = self.module_list[i]
            try:
                batch_normalize = int(self.blocks[i+1]["batch_normalize"])
            except:
                batch_normalize = 0

            conv = model[0]


            if (batch_normalize):
                bn = model[1]

                #Get the number of weights of Batch Norm Layer
                num_bn_biases = bn.bias.numel()
```

```
                    #Load the weights
                    bn_biases = torch.from_numpy(weights[ptr:ptr +␣
↪num_bn_biases])
                    ptr += num_bn_biases

                    bn_weights = torch.from_numpy(weights[ptr: ptr +␣
↪num_bn_biases])
                    ptr  += num_bn_biases

                    bn_running_mean = torch.from_numpy(weights[ptr: ptr +␣
↪num_bn_biases])
                    ptr  += num_bn_biases

                    bn_running_var = torch.from_numpy(weights[ptr: ptr +␣
↪num_bn_biases])
                    ptr  += num_bn_biases

                    #Cast the loaded weights into dims of model weights.
                    bn_biases = bn_biases.view_as(bn.bias.data)
                    bn_weights = bn_weights.view_as(bn.weight.data)
                    bn_running_mean = bn_running_mean.view_as(bn.running_mean)
                    bn_running_var = bn_running_var.view_as(bn.running_var)

                    #Copy the data to model
                    bn.bias.data.copy_(bn_biases)
                    bn.weight.data.copy_(bn_weights)
                    bn.running_mean.copy_(bn_running_mean)
                    bn.running_var.copy_(bn_running_var)

                else:
                    #Number of biases
                    num_biases = conv.bias.numel()

                    #Load the weights
                    conv_biases = torch.from_numpy(weights[ptr: ptr +␣
↪num_biases])
                    ptr = ptr + num_biases

                    #reshape the loaded weights according to the dims of the␣
↪model weights
                    conv_biases = conv_biases.view_as(conv.bias.data)

                    #Finally copy the data
                    conv.bias.data.copy_(conv_biases)
```

```python
                #Let us load the weights for the Convolutional layers
                num_weights = conv.weight.numel()

                #Do the same as above for weights
                conv_weights = torch.from_numpy(weights[ptr:ptr+num_weights])
                ptr = ptr + num_weights

                conv_weights = conv_weights.view_as(conv.weight.data)
                conv.weight.data.copy_(conv_weights)
```

[ ]:
```python
# utils.py
from __future__ import division

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import numpy as np
import cv2

def unique(tensor):
    tensor_np = tensor.cpu().numpy()
    unique_np = np.unique(tensor_np)
    unique_tensor = torch.from_numpy(unique_np)

    tensor_res = tensor.new(unique_tensor.shape)
    tensor_res.copy_(unique_tensor)
    return tensor_res


def bbox_iou(box1, box2):
    """
    Returns the IoU of two bounding boxes


    """
    #Get the coordinates of bounding boxes
    b1_x1, b1_y1, b1_x2, b1_y2 = box1[:,0], box1[:,1], box1[:,2], box1[:,3]
    b2_x1, b2_y1, b2_x2, b2_y2 = box2[:,0], box2[:,1], box2[:,2], box2[:,3]

    #get the corrdinates of the intersection rectangle
    inter_rect_x1 =  torch.max(b1_x1, b2_x1)
    inter_rect_y1 =  torch.max(b1_y1, b2_y1)
    inter_rect_x2 =  torch.min(b1_x2, b2_x2)
    inter_rect_y2 =  torch.min(b1_y2, b2_y2)

    #Intersection area
```

```
    inter_area = torch.clamp(inter_rect_x2 - inter_rect_x1 + 1, min=0) * torch.
↪clamp(inter_rect_y2 - inter_rect_y1 + 1, min=0)

    #Union Area
    b1_area = (b1_x2 - b1_x1 + 1)*(b1_y2 - b1_y1 + 1)
    b2_area = (b2_x2 - b2_x1 + 1)*(b2_y2 - b2_y1 + 1)

    iou = inter_area / (b1_area + b2_area - inter_area)

    return iou

def predict_transform(prediction, inp_dim, anchors, num_classes, CUDA=True):
    """Transform predictions from feature maps to bounding boxes"""
    batch_size = prediction.size(0)
    stride = inp_dim // prediction.size(2)
    grid_size = inp_dim // stride
    bbox_attrs = 5 + num_classes
    num_anchors = len(anchors)

    # Reshape prediction to [batch_size, bbox_attrs * num_anchors, grid_size *␣
↪grid_size]
    prediction = prediction.view(batch_size, bbox_attrs * num_anchors,␣
↪grid_size * grid_size)
    prediction = prediction.transpose(1, 2).contiguous()
    prediction = prediction.view(batch_size, grid_size * grid_size *␣
↪num_anchors, bbox_attrs)

    # Scale anchors by stride
    anchors = [(a[0]/stride, a[1]/stride) for a in anchors]

    # Sigmoid the center_X, center_Y. and object confidence
    prediction[:, :, 0] = torch.sigmoid(prediction[:, :, 0])
    prediction[:, :, 1] = torch.sigmoid(prediction[:, :, 1])
    prediction[:, :, 4] = torch.sigmoid(prediction[:, :, 4])

    # Add the center offsets
    device = prediction.device
    grid = torch.arange(grid_size, device=device)
    a, b = torch.meshgrid(grid, grid, indexing='ij')
    x_offset = a.contiguous().view(-1, 1)
    y_offset = b.contiguous().view(-1, 1)

    x_y_offset = torch.cat((x_offset, y_offset), 1).repeat(1, num_anchors).
↪view(-1, 2).unsqueeze(0)
    prediction[:, :, :2] += x_y_offset

    # Log space transform height and the width
```

```
    anchors = torch.FloatTensor(anchors).to(device)
    anchors = anchors.repeat(grid_size * grid_size, 1).unsqueeze(0)
    prediction[:, :, 2:4] = torch.exp(prediction[:, :, 2:4]) * anchors

    # Sigmoid the class scores
    prediction[:, :, 5:5+num_classes] = torch.sigmoid(prediction[:, :, 5:
↪5+num_classes])

    # Resize the detection map to the input image size
    prediction[:, :, :4] *= stride

    return prediction

def write_results(prediction, confidence, num_classes, nms_conf = 0.4):
    conf_mask = (prediction[:,:,4] > confidence).float().unsqueeze(2)
    prediction = prediction*conf_mask

    box_corner = prediction.new(prediction.shape)
    box_corner[:,:,0] = (prediction[:,:,0] - prediction[:,:,2]/2)
    box_corner[:,:,1] = (prediction[:,:,1] - prediction[:,:,3]/2)
    box_corner[:,:,2] = (prediction[:,:,0] + prediction[:,:,2]/2)
    box_corner[:,:,3] = (prediction[:,:,1] + prediction[:,:,3]/2)
    prediction[:,:,:4] = box_corner[:,:,:4]

    batch_size = prediction.size(0)

    write = False



    for ind in range(batch_size):
        image_pred = prediction[ind]          #image Tensor
       #confidence threshholding
       #NMS

        max_conf, max_conf_score = torch.max(image_pred[:,5:5+ num_classes], 1)
        max_conf = max_conf.float().unsqueeze(1)
        max_conf_score = max_conf_score.float().unsqueeze(1)
        seq = (image_pred[:,:5], max_conf, max_conf_score)
        image_pred = torch.cat(seq, 1)

        non_zero_ind =  (torch.nonzero(image_pred[:,4]))
        try:
            image_pred_ = image_pred[non_zero_ind.squeeze(),:].view(-1,7)
        except:
            continue
```

```
        if image_pred_.shape[0] == 0:
            continue
#

        #Get the various classes detected in the image
        img_classes = unique(image_pred_[:,-1])  # -1 index holds the class␣
↪index


        for cls in img_classes:
            #perform NMS


            #get the detections with one particular class
            cls_mask = image_pred_*(image_pred_[:,-1] == cls).float().
↪unsqueeze(1)
            class_mask_ind = torch.nonzero(cls_mask[:,-2]).squeeze()
            image_pred_class = image_pred_[class_mask_ind].view(-1,7)

            #sort the detections such that the entry with the maximum objectness
            #confidence is at the top
            conf_sort_index = torch.sort(image_pred_class[:,4], descending =␣
↪True )[1]
            image_pred_class = image_pred_class[conf_sort_index]
            idx = image_pred_class.size(0)   #Number of detections

            for i in range(idx):
                #Get the IOUs of all boxes that come after the one we are␣
↪looking at
                #in the loop
                try:
                    ious = bbox_iou(image_pred_class[i].unsqueeze(0),␣
↪image_pred_class[i+1:])
                except ValueError:
                    break

                except IndexError:
                    break

                #Zero out all the detections that have IoU > treshhold
                iou_mask = (ious < nms_conf).float().unsqueeze(1)
                image_pred_class[i+1:] *= iou_mask

                #Remove the non-zero entries
                non_zero_ind = torch.nonzero(image_pred_class[:,4]).squeeze()
                image_pred_class = image_pred_class[non_zero_ind].view(-1,7)
```

```
            batch_ind = image_pred_class.new(image_pred_class.size(0), 1).
↪fill_(ind)      #Repeat the batch_id for as many detections of the class cls␣
↪in the image
            seq = batch_ind, image_pred_class

            if not write:
                output = torch.cat(seq,1)
                write = True
            else:
                out = torch.cat(seq,1)
                output = torch.cat((output,out))

    try:
        return output
    except:
        return 0

def letterbox_image(img, inp_dim):
    '''resize image with unchanged aspect ratio using padding'''
    img_w, img_h = img.shape[1], img.shape[0]
    w, h = inp_dim
    new_w = int(img_w * min(w/img_w, h/img_h))
    new_h = int(img_h * min(w/img_w, h/img_h))
    resized_image = cv2.resize(img, (new_w,new_h), interpolation = cv2.
↪INTER_CUBIC)

    canvas = np.full((inp_dim[1], inp_dim[0], 3), 128)

    canvas[(h-new_h)//2:(h-new_h)//2 + new_h,(w-new_w)//2:(w-new_w)//2 + new_w,␣
↪ :] = resized_image

    return canvas

def prep_image(img, inp_dim):
    """
    Prepare image for inputting to the neural network.

    Returns a Variable
    """
    img = (letterbox_image(img, (inp_dim, inp_dim)))
    img = img[:,:,::-1].transpose((2,0,1)).copy()
    img = torch.from_numpy(img).float().div(255.0).unsqueeze(0)
    return img

def load_classes(namesfile):
    fp = open(namesfile, "r")
    names = fp.read().split("\n")[:-1]
```

```
    return names
```

```
[ ]: # yolov4.py

import torch
import torch.nn as nn
import torch.nn.functional as F
import math
import numpy as np

def bbox_ciou(box1, box2, x1y1x2y2=True):
    """
    Calculate CIoU loss between two bounding boxes
    """
    if not x1y1x2y2:
        # Convert nx4 boxes from [x, y, w, h] to [x1, y1, x2, y2]
        b1_x1, b1_x2 = box1[:, 0] - box1[:, 2] / 2, box1[:, 0] + box1[:, 2] / 2
        b1_y1, b1_y2 = box1[:, 1] - box1[:, 3] / 2, box1[:, 1] + box1[:, 3] / 2
        b2_x1, b2_x2 = box2[:, 0] - box2[:, 2] / 2, box2[:, 0] + box2[:, 2] / 2
        b2_y1, b2_y2 = box2[:, 1] - box2[:, 3] / 2, box2[:, 1] + box2[:, 3] / 2
    else:
        # Get the coordinates of bounding boxes
        b1_x1, b1_y1, b1_x2, b1_y2 = box1[:, 0], box1[:, 1], box1[:, 2], box1[:
↪, 3]
        b2_x1, b2_y1, b2_x2, b2_y2 = box2[:, 0], box2[:, 1], box2[:, 2], box2[:
↪, 3]

    # Intersection area
    inter = (torch.min(b1_x2, b2_x2) - torch.max(b1_x1, b2_x1)).clamp(0) * \
            (torch.min(b1_y2, b2_y2) - torch.max(b1_y1, b2_y1)).clamp(0)

    # Union Area
    w1, h1 = b1_x2 - b1_x1, b1_y2 - b1_y1
    w2, h2 = b2_x2 - b2_x1, b2_y2 - b2_y1
    union = w1 * h1 + w2 * h2 - inter + 1e-16

    iou = inter / union

    # Get enclosed coordinates
    cw = torch.max(b1_x2, b2_x2) - torch.min(b1_x1, b2_x1)
    ch = torch.max(b1_y2, b2_y2) - torch.min(b1_y1, b2_y1)

    # Get diagonal distance
    c2 = cw ** 2 + ch ** 2 + 1e-16

    # Get center distance
    rho2 = ((b2_x1 + b2_x2 - b1_x1 - b1_x2) ** 2 +
```

```python
            (b2_y1 + b2_y2 - b1_y1 - b1_y2) ** 2) / 4

    # Calculate v and alpha
    v = (4 / (math.pi ** 2)) * torch.pow(torch.atan(w2 / (h2 + 1e-16)) - torch.
↪atan(w1 / (h1 + 1e-16)), 2)
    with torch.no_grad():
        alpha = v / (v - iou + (1 + 1e-16))

    # CIoU
    return iou - (rho2 / c2 + v * alpha)

def xywh2xyxy(x):
    # Convert bounding box format from [x, y, w, h] to [x1, y1, x2, y2]
    y = x.clone()
    y[:, 0] = x[:, 0] - x[:, 2] / 2
    y[:, 1] = x[:, 1] - x[:, 3] / 2
    y[:, 2] = x[:, 0] + x[:, 2] / 2
    y[:, 3] = x[:, 1] + x[:, 3] / 2
    return y

class YOLOv4Loss(nn.Module):
    def __init__(self, anchors, num_classes=80):
        super(YOLOv4Loss, self).__init__()
        self.anchors = anchors
        self.num_classes = num_classes
        self.ignore_thres = 0.5
        self.obj_scale = 1
        self.noobj_scale = 100

    def forward(self, predictions, targets):
        """
        Args:
            predictions: tensor of shape (batch_size, num_boxes, 5 +␣
↪num_classes)
            targets: list of dicts containing boxes and labels
        """
        device = predictions.device
        batch_size = predictions.size(0)
        total_loss = torch.tensor(0., requires_grad=True, device=device)

        for i in range(batch_size):
            pred = predictions[i]
            target = targets[i]

            # Get target boxes and labels and move them to the correct device
            target_boxes = target['boxes'].to(device)
            target_labels = target['labels'].to(device)
```

```python
            if len(target_boxes) == 0:
                continue

            # Get prediction components
            pred_boxes = pred[..., :4]  # [x, y, w, h]
            pred_conf = pred[..., 4]    # objectness
            pred_cls = pred[..., 5:]    # class scores

            # Calculate IoU for each predicted box with each target box
            num_pred = pred_boxes.size(0)
            num_target = target_boxes.size(0)

            # Expand dimensions for broadcasting
            pred_boxes = pred_boxes.unsqueeze(1).repeat(1, num_target, 1)
            target_boxes = target_boxes.unsqueeze(0).repeat(num_pred, 1, 1)

            # Calculate CIoU loss
            ciou = bbox_ciou(pred_boxes.view(-1, 4), target_boxes.view(-1, 4))
            ciou = ciou.view(num_pred, num_target)

            # For each target, find the best matching prediction
            best_ious, best_idx = ciou.max(dim=0)

            # Calculate box loss using CIoU
            box_loss = (1.0 - best_ious).mean()

            # Calculate objectness loss
            obj_mask = torch.zeros_like(pred_conf)
            obj_mask[best_idx] = 1
            obj_loss = F.binary_cross_entropy_with_logits(pred_conf, obj_mask)

            # Calculate classification loss
            target_cls = torch.zeros_like(pred_cls)
            for j, label in enumerate(target_labels):
                target_cls[best_idx[j], label] = 1
            cls_loss = F.binary_cross_entropy_with_logits(pred_cls, target_cls)

            # Combine losses
            batch_loss = box_loss + obj_loss + cls_loss
            total_loss = total_loss + batch_loss

        return total_loss / batch_size

    def __call__(self, predictions, targets):
        return self.forward(predictions, targets)
```

[ ]:
```python
# metrics.py
import numpy as np
import torch
from collections import defaultdict
from pycocotools.coco import COCO
from pycocotools.cocoeval import COCOeval

def compute_ap(recall, precision):
    """Compute the average precision, given the recall and precision curves."""
    # Append sentinel values to beginning and end
    mrec = np.concatenate(([0.], recall, [1.]))
    mpre = np.concatenate(([0.], precision, [0.]))

    # Compute the precision envelope
    for i in range(mpre.size - 1, 0, -1):
        mpre[i - 1] = np.maximum(mpre[i - 1], mpre[i])

    # Create a list of indexes where the recall changes
    i = np.where(mrec[1:] != mrec[:-1])[0]

    # Calculate the area under PR curve by sum of rectangular blocks
    ap = np.sum((mrec[i + 1] - mrec[i]) * mpre[i + 1])
    return ap

def bbox_iou(box1, box2, x1y1x2y2=True):
    """Returns the IoU of two bounding boxes."""
    if not x1y1x2y2:
        # Transform from center and width to exact coordinates
        b1_x1, b1_x2 = box1[:, 0] - box1[:, 2] / 2, box1[:, 0] + box1[:, 2] / 2
        b1_y1, b1_y2 = box1[:, 1] - box1[:, 3] / 2, box1[:, 1] + box1[:, 3] / 2
        b2_x1, b2_x2 = box2[:, 0] - box2[:, 2] / 2, box2[:, 0] + box2[:, 2] / 2
        b2_y1, b2_y2 = box2[:, 1] - box2[:, 3] / 2, box2[:, 1] + box2[:, 3] / 2
    else:
        # Get the coordinates of bounding boxes
        b1_x1, b1_y1, b1_x2, b1_y2 = box1[:, 0], box1[:, 1], box1[:, 2], box1[:
↪, 3]
        b2_x1, b2_y1, b2_x2, b2_y2 = box2[:, 0], box2[:, 1], box2[:, 2], box2[:
↪, 3]

    # Get the coordinates of the intersection rectangle
    inter_rect_x1 = torch.max(b1_x1, b2_x1)
    inter_rect_y1 = torch.max(b1_y1, b2_y1)
    inter_rect_x2 = torch.min(b1_x2, b2_x2)
    inter_rect_y2 = torch.min(b1_y2, b2_y2)

    # Intersection area
```

```
    inter_area = torch.clamp(inter_rect_x2 - inter_rect_x1, 0) * torch.
↪clamp(inter_rect_y2 - inter_rect_y1, 0)

    # Union Area
    b1_area = (b1_x2 - b1_x1) * (b1_y2 - b1_y1)
    b2_area = (b2_x2 - b2_x1) * (b2_y2 - b2_y1)

    return inter_area / (b1_area + b2_area - inter_area + 1e-16)

def evaluate_coco_map(model, data_loader, coco_gt):
    """Evaluate mAP on COCO validation set"""
    model.eval()
    device = next(model.parameters()).device

    # Prepare for COCO evaluation
    coco_dt = []
    image_ids = []

    with torch.no_grad():
        for imgs, targets in data_loader:
            imgs = imgs.to(device)
            batch_size = imgs.shape[0]

            # Forward pass with CUDA flag
            predictions = model(imgs, device == torch.device("cuda"))

            # Process predictions
            for i in range(batch_size):
                img_id = targets[i]['image_id'].item()
                image_ids.append(img_id)

                if len(predictions[i]) == 0:
                    continue

                # Convert predictions to COCO format
                for pred in predictions[i]:
                    x1, y1, x2, y2 = [p.item() for p in pred[:4]]
                    conf = pred[4].item()
                    cls_conf = pred[5].item()
                    cls_pred = int(pred[6].item())  # Convert class prediction␣
↪to integer

                    coco_dt.append({
                        'image_id': img_id,
                        'category_id': cls_pred,
                        'bbox': [float(x1), float(y1), float(x2 - x1), float(y2␣
↪- y1)],
```

```
                    'score': float(conf * cls_conf)
                })

    if len(coco_dt) == 0:
        return 0.0

    # Save predictions to temporary file
    _, tmp_file = tempfile.mkstemp()
    with open(tmp_file, 'w') as f:
        json.dump(coco_dt, f)

    # Load predictions in COCO format
    coco_pred = coco_gt.loadRes(tmp_file)

    # Run COCO evaluation
    coco_eval = COCOeval(coco_gt, coco_pred, 'bbox')
    coco_eval.params.imgIds = image_ids
    coco_eval.evaluate()
    coco_eval.accumulate()
    coco_eval.summarize()

    return coco_eval.stats[0]  # Return mAP@[0.5:0.95]
```

1.0.4 Task 2: Training

1. Train the YOLOv4 model on the COCO dataset (or another dataset if you have one available). Here the purpose is not to get the best possible model (that would require implementing all of the "bag of freebies" training tricks described in the paper), but just some of them, to get a feel for their importance. - Done
2. Get a set of ImageNet pretrained weights for CSPDarknet53 from the Darknet GitHub repository - Done
3. Add a method to load the pretrained weights into the backbone portion of your PyTorch YOLOv4 model. - Done
4. Implement a basic `train_yolo` function similar to the `train_model` function you developed in previous labs for classifiers that preprocesses the input with basic augmentation transformations, converts the anchor-relative outputs to bounding box coordinates, computes MSE loss for the bounding box coordinates, backpropagates the loss, and takes a step for the optimizer. Use the recommended IoU thresholds to determine which predicted bounding boxes to include in the loss. You will find many examples of how to do this online. - Done
5. Train your model on COCO. Training on the full dataset to completion would take several days, so you can stop early after verifying the model is learning in the first few epochs. - Done, trained only for 1 epoch
6. Compute mAP for your model on the COCO validation set. - Done
7. Implement the CIoU loss function and observe its effect on mAP. - Done, can be found under model/ciou.py file
8. (Optional) Train on COCO to completion and see how close you can get to the mAP reported in the paper.

I was using yolo pretrained weights with one epoch fine tune on coco dataset (due to time limitations), used 4 batch size based on gpu allowance. Samely, I trained from scratch only 1 epoch, the inference results of which are not good. All information regarding the training can be found in logs folder.

[ ]:
```
# # # train_from_scratch.log

# jupyter-st125457@puffer:~/rtml/a2_yolov4$ python train.py --data ./data/coco.
↪yaml --batch-size 4
# Loading YOLOv4 Model...
# No pretrained weights found. Training from scratch...
# Loading data configuration...

# Creating datasets...
# Training data: data/coco/val2017
# Training annotations: data/coco/annotations/instances_val2017.json
# Validation data: data/coco/val2017
# Validation annotations: data/coco/annotations/instances_val2017.json
# loading annotations into memory...
# Done (t=0.49s)
# creating index...
# index created!
# loading annotations into memory...
# Done (t=0.51s)
# creating index...
# index created!
# Dataset size: 5000 training images, 5000 validation images
# loading annotations into memory...
# Done (t=0.45s)
# creating index...
# index created!

# Starting training...
# Epoch: 0, Batch: 0, Loss: 2.4824
# Epoch: 0, Batch: 10, Loss: 2.5740
# Epoch: 0, Batch: 20, Loss: 2.5746
# Epoch: 0, Batch: 30, Loss: 2.5707
# Epoch: 0, Batch: 40, Loss: 2.5751
# Epoch: 0, Batch: 50, Loss: 2.5773
# Epoch: 0, Batch: 60, Loss: 2.5806
# Epoch: 0, Batch: 70, Loss: 2.5834
# Epoch: 0, Batch: 80, Loss: 2.5784
# Epoch: 0, Batch: 90, Loss: 2.5810
# Epoch: 0, Batch: 100, Loss: 2.5752
# Epoch: 0, Batch: 110, Loss: 2.5788
# Epoch: 0, Batch: 120, Loss: 2.5766
# Epoch: 0, Batch: 130, Loss: 2.5735
```

```
# Epoch: 0, Batch: 140, Loss: 2.5740
# Epoch: 0, Batch: 150, Loss: 2.5723
# Epoch: 0, Batch: 160, Loss: 2.5729
# Epoch: 0, Batch: 170, Loss: 2.5702
# Epoch: 0, Batch: 180, Loss: 2.5639
# Epoch: 0, Batch: 190, Loss: 2.5637
# Epoch: 0, Batch: 200, Loss: 2.5678
# Epoch: 0, Batch: 210, Loss: 2.5691
# Epoch: 0, Batch: 220, Loss: 2.5682
# Epoch: 0, Batch: 230, Loss: 2.5680
# Epoch: 0, Batch: 240, Loss: 2.5668
# Epoch: 0, Batch: 250, Loss: 2.5660
# Epoch: 0, Batch: 260, Loss: 2.5679
# Epoch: 0, Batch: 270, Loss: 2.5664
# Epoch: 0, Batch: 280, Loss: 2.5651
# Epoch: 0, Batch: 290, Loss: 2.5668
# Epoch: 0, Batch: 300, Loss: 2.5665
# Epoch: 0, Batch: 310, Loss: 2.5661
# Epoch: 0, Batch: 320, Loss: 2.5646
# Epoch: 0, Batch: 330, Loss: 2.5626
# Epoch: 0, Batch: 340, Loss: 2.5635
# Epoch: 0, Batch: 350, Loss: 2.5646
# Epoch: 0, Batch: 360, Loss: 2.5651
# Epoch: 0, Batch: 370, Loss: 2.5667
# Epoch: 0, Batch: 380, Loss: 2.5682
# Epoch: 0, Batch: 390, Loss: 2.5681
# Epoch: 0, Batch: 400, Loss: 2.5673
# Epoch: 0, Batch: 410, Loss: 2.5671
# Epoch: 0, Batch: 420, Loss: 2.5642
# Epoch: 0, Batch: 430, Loss: 2.5632
# Epoch: 0, Batch: 440, Loss: 2.5626
# Epoch: 0, Batch: 450, Loss: 2.5623
# Epoch: 0, Batch: 460, Loss: 2.5631
# Epoch: 0, Batch: 470, Loss: 2.5619
# Epoch: 0, Batch: 480, Loss: 2.5620
# Epoch: 0, Batch: 490, Loss: 2.5628
# Epoch: 0, Batch: 500, Loss: 2.5623
# Epoch: 0, Batch: 510, Loss: 2.5630
# Epoch: 0, Batch: 520, Loss: 2.5617
# Epoch: 0, Batch: 530, Loss: 2.5620
# Epoch: 0, Batch: 540, Loss: 2.5610
# Epoch: 0, Batch: 550, Loss: 2.5609
# Epoch: 0, Batch: 560, Loss: 2.5611
# Epoch: 0, Batch: 570, Loss: 2.5622
# Epoch: 0, Batch: 580, Loss: 2.5628
# Epoch: 0, Batch: 590, Loss: 2.5631
# Epoch: 0, Batch: 600, Loss: 2.5616
```

```
# Epoch: 0, Batch: 610, Loss: 2.5613
# Epoch: 0, Batch: 620, Loss: 2.5622
# Epoch: 0, Batch: 630, Loss: 2.5628
# Epoch: 0, Batch: 640, Loss: 2.5643
# Epoch: 0, Batch: 650, Loss: 2.5650
# Epoch: 0, Batch: 660, Loss: 2.5645
# Epoch: 0, Batch: 670, Loss: 2.5628
# Epoch: 0, Batch: 680, Loss: 2.5631
# Epoch: 0, Batch: 690, Loss: 2.5633
# Epoch: 0, Batch: 700, Loss: 2.5632
# Epoch: 0, Batch: 710, Loss: 2.5623
# Epoch: 0, Batch: 720, Loss: 2.5627
# Epoch: 0, Batch: 730, Loss: 2.5627
# Epoch: 0, Batch: 740, Loss: 2.5628
# Epoch: 0, Batch: 750, Loss: 2.5633
# Epoch: 0, Batch: 760, Loss: 2.5631
# Epoch: 0, Batch: 770, Loss: 2.5633
# Epoch: 0, Batch: 780, Loss: 2.5626
# Epoch: 0, Batch: 790, Loss: 2.5614
# Epoch: 0, Batch: 800, Loss: 2.5614
# Epoch: 0, Batch: 810, Loss: 2.5612
# Epoch: 0, Batch: 820, Loss: 2.5618
# Epoch: 0, Batch: 830, Loss: 2.5619
# Epoch: 0, Batch: 840, Loss: 2.5606
# Epoch: 0, Batch: 850, Loss: 2.5610
# Epoch: 0, Batch: 860, Loss: 2.5601
# Epoch: 0, Batch: 870, Loss: 2.5600
# Epoch: 0, Batch: 880, Loss: 2.5600
# Epoch: 0, Batch: 890, Loss: 2.5605
# Epoch: 0, Batch: 900, Loss: 2.5606
# Epoch: 0, Batch: 910, Loss: 2.5601
# Epoch: 0, Batch: 920, Loss: 2.5590
# Epoch: 0, Batch: 930, Loss: 2.5601
# Epoch: 0, Batch: 940, Loss: 2.5608
# Epoch: 0, Batch: 950, Loss: 2.5608
# Epoch: 0, Batch: 960, Loss: 2.5608
# Epoch: 0, Batch: 970, Loss: 2.5605
# Epoch: 0, Batch: 980, Loss: 2.5608
# Epoch: 0, Batch: 990, Loss: 2.5604
# Epoch: 0, Batch: 1000, Loss: 2.5607
# Epoch: 0, Batch: 1010, Loss: 2.5610
# Epoch: 0, Batch: 1020, Loss: 2.5609
# Epoch: 0, Batch: 1030, Loss: 2.5605
# Epoch: 0, Batch: 1040, Loss: 2.5598
# Epoch: 0, Batch: 1050, Loss: 2.5607
# Epoch: 0, Batch: 1060, Loss: 2.5611
# Epoch: 0, Batch: 1070, Loss: 2.5610
```

```
# Epoch: 0, Batch: 1080, Loss: 2.5608
# Epoch: 0, Batch: 1090, Loss: 2.5598
# Epoch: 0, Batch: 1100, Loss: 2.5600
# Epoch: 0, Batch: 1110, Loss: 2.5602
# Epoch: 0, Batch: 1120, Loss: 2.5603
# Epoch: 0, Batch: 1130, Loss: 2.5599
# Epoch: 0, Batch: 1140, Loss: 2.5597
# Epoch: 0, Batch: 1150, Loss: 2.5604
# Epoch: 0, Batch: 1160, Loss: 2.5607
# Epoch: 0, Batch: 1170, Loss: 2.5612
# Epoch: 0, Batch: 1180, Loss: 2.5612
# Epoch: 0, Batch: 1190, Loss: 2.5607
# Epoch: 0, Batch: 1200, Loss: 2.5608
# Epoch: 0, Batch: 1210, Loss: 2.5614
# Epoch: 0, Batch: 1220, Loss: 2.5616
# Epoch: 0, Batch: 1230, Loss: 2.5614
# Epoch: 0, Batch: 1240, Loss: 2.5613

# Evaluating mAP...

# Loading and preparing results...
# DONE (t=7.37s)
# creating index...
# index created!
# Running per image evaluation...
# Evaluate annotation type *bbox*
# DONE (t=2.92s).
# Accumulating evaluation results...
# DONE (t=0.11s).
#  Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.
 ↪000
#  Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.
 ↪000
#  Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.
 ↪000
#  Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.
 ↪000
#  Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.
 ↪000
#  Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.
 ↪000
#  Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.
 ↪000
#  Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.
 ↪000
```

```
#  Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.
↪000
#  Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.
↪000
#  Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.
↪000
#  Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.
↪000
# Epoch 0 mAP: 0.0000
```

[ ]:
```
# # train_pretrained.log

# jupyter-st125457@puffer:~/rtml/a2_yolov4$ python train.py --data ./data/coco.
↪yaml --batch-size 4 --weights checkpoints/yolov4.weights
# Loading YOLOv4 Model...
# Loading weights from weights/yolov4.weights
# Loading data configuration...

# Creating datasets...
# Training data: data/coco/val2017
# Training annotations: data/coco/annotations/instances_val2017.json
# Validation data: data/coco/val2017
# Validation annotations: data/coco/annotations/instances_val2017.json
# loading annotations into memory...
# Done (t=0.47s)
# creating index...
# index created!
# loading annotations into memory...
# Done (t=0.46s)
# creating index...
# index created!
# Dataset size: 5000 training images, 5000 validation images
# loading annotations into memory...
# Done (t=0.45s)
# creating index...
# index created!

# Starting training...
# Epoch: 0, Batch: 0, Loss: 2.2426
# Epoch: 0, Batch: 10, Loss: 2.2749
# Epoch: 0, Batch: 20, Loss: 2.2745
# Epoch: 0, Batch: 30, Loss: 2.2340
# Epoch: 0, Batch: 40, Loss: 2.2419
# Epoch: 0, Batch: 50, Loss: 2.2415
# Epoch: 0, Batch: 60, Loss: 2.2394
# Epoch: 0, Batch: 70, Loss: 2.2434
# Epoch: 0, Batch: 80, Loss: 2.2408
```

```
# Epoch: 0, Batch: 90, Loss: 2.2423
# Epoch: 0, Batch: 100, Loss: 2.2455
# Epoch: 0, Batch: 110, Loss: 2.2412
# Epoch: 0, Batch: 120, Loss: 2.2412
# Epoch: 0, Batch: 130, Loss: 2.2417
# Epoch: 0, Batch: 140, Loss: 2.2451
# Epoch: 0, Batch: 150, Loss: 2.2478
# Epoch: 0, Batch: 160, Loss: 2.2490
# Epoch: 0, Batch: 170, Loss: 2.2484
# Epoch: 0, Batch: 180, Loss: 2.2448
# Epoch: 0, Batch: 190, Loss: 2.2456
# Epoch: 0, Batch: 200, Loss: 2.2422
# Epoch: 0, Batch: 210, Loss: 2.2430
# Epoch: 0, Batch: 220, Loss: 2.2419
# Epoch: 0, Batch: 230, Loss: 2.2410
# Epoch: 0, Batch: 240, Loss: 2.2396
# Epoch: 0, Batch: 250, Loss: 2.2406
# Epoch: 0, Batch: 260, Loss: 2.2390
# Epoch: 0, Batch: 270, Loss: 2.2382
# Epoch: 0, Batch: 280, Loss: 2.2385
# Epoch: 0, Batch: 290, Loss: 2.2383
# Epoch: 0, Batch: 300, Loss: 2.2387
# Epoch: 0, Batch: 310, Loss: 2.2378
# Epoch: 0, Batch: 320, Loss: 2.2390
# Epoch: 0, Batch: 330, Loss: 2.2384
# Epoch: 0, Batch: 340, Loss: 2.2388
# Epoch: 0, Batch: 350, Loss: 2.2392
# Epoch: 0, Batch: 360, Loss: 2.2409
# Epoch: 0, Batch: 370, Loss: 2.2418
# Epoch: 0, Batch: 380, Loss: 2.2414
# Epoch: 0, Batch: 390, Loss: 2.2419
# Epoch: 0, Batch: 400, Loss: 2.2430
# Epoch: 0, Batch: 410, Loss: 2.2438
# Epoch: 0, Batch: 420, Loss: 2.2445
# Epoch: 0, Batch: 430, Loss: 2.2446
# Epoch: 0, Batch: 440, Loss: 2.2453
# Epoch: 0, Batch: 450, Loss: 2.2456
# Epoch: 0, Batch: 460, Loss: 2.2462
# Epoch: 0, Batch: 470, Loss: 2.2467
# Epoch: 0, Batch: 480, Loss: 2.2472
# Epoch: 0, Batch: 490, Loss: 2.2481
# Epoch: 0, Batch: 500, Loss: 2.2465
# Epoch: 0, Batch: 510, Loss: 2.2463
# Epoch: 0, Batch: 520, Loss: 2.2469
# Epoch: 0, Batch: 530, Loss: 2.2470
# Epoch: 0, Batch: 540, Loss: 2.2471
# Epoch: 0, Batch: 550, Loss: 2.2469
```

```
# Epoch: 0, Batch: 560, Loss: 2.2455
# Epoch: 0, Batch: 570, Loss: 2.2462
# Epoch: 0, Batch: 580, Loss: 2.2446
# Epoch: 0, Batch: 590, Loss: 2.2453
# Epoch: 0, Batch: 600, Loss: 2.2461
# Epoch: 0, Batch: 610, Loss: 2.2452
# Epoch: 0, Batch: 620, Loss: 2.2434
# Epoch: 0, Batch: 630, Loss: 2.2432
# Epoch: 0, Batch: 640, Loss: 2.2434
# Epoch: 0, Batch: 650, Loss: 2.2437
# Epoch: 0, Batch: 660, Loss: 2.2437
# Epoch: 0, Batch: 670, Loss: 2.2436
# Epoch: 0, Batch: 680, Loss: 2.2436
# Epoch: 0, Batch: 690, Loss: 2.2434
# Epoch: 0, Batch: 700, Loss: 2.2441
# Epoch: 0, Batch: 710, Loss: 2.2445
# Epoch: 0, Batch: 720, Loss: 2.2454
# Epoch: 0, Batch: 730, Loss: 2.2455
# Epoch: 0, Batch: 740, Loss: 2.2453
# Epoch: 0, Batch: 750, Loss: 2.2441
# Epoch: 0, Batch: 760, Loss: 2.2447
# Epoch: 0, Batch: 770, Loss: 2.2449
# Epoch: 0, Batch: 780, Loss: 2.2448
# Epoch: 0, Batch: 790, Loss: 2.2440
# Epoch: 0, Batch: 800, Loss: 2.2432
# Epoch: 0, Batch: 810, Loss: 2.2431
# Epoch: 0, Batch: 820, Loss: 2.2426
# Epoch: 0, Batch: 830, Loss: 2.2418
# Epoch: 0, Batch: 840, Loss: 2.2418
# Epoch: 0, Batch: 850, Loss: 2.2423
# Epoch: 0, Batch: 860, Loss: 2.2429
# Epoch: 0, Batch: 870, Loss: 2.2424
# Epoch: 0, Batch: 880, Loss: 2.2426
# Epoch: 0, Batch: 890, Loss: 2.2429
# Epoch: 0, Batch: 900, Loss: 2.2429
# Epoch: 0, Batch: 910, Loss: 2.2428
# Epoch: 0, Batch: 920, Loss: 2.2432
# Epoch: 0, Batch: 930, Loss: 2.2420
# Epoch: 0, Batch: 940, Loss: 2.2422
# Epoch: 0, Batch: 950, Loss: 2.2424
# Epoch: 0, Batch: 960, Loss: 2.2421
# Epoch: 0, Batch: 970, Loss: 2.2426
# Epoch: 0, Batch: 980, Loss: 2.2430
# Epoch: 0, Batch: 990, Loss: 2.2435
# Epoch: 0, Batch: 1000, Loss: 2.2431
# Epoch: 0, Batch: 1010, Loss: 2.2430
# Epoch: 0, Batch: 1020, Loss: 2.2433
```

```
# Epoch: 0, Batch: 1030, Loss: 2.2440
# Epoch: 0, Batch: 1040, Loss: 2.2443
# Epoch: 0, Batch: 1050, Loss: 2.2447
# Epoch: 0, Batch: 1060, Loss: 2.2443
# Epoch: 0, Batch: 1070, Loss: 2.2446
# Epoch: 0, Batch: 1080, Loss: 2.2448
# Epoch: 0, Batch: 1090, Loss: 2.2450
# Epoch: 0, Batch: 1100, Loss: 2.2454
# Epoch: 0, Batch: 1110, Loss: 2.2452
# Epoch: 0, Batch: 1120, Loss: 2.2449
# Epoch: 0, Batch: 1130, Loss: 2.2453
# Epoch: 0, Batch: 1140, Loss: 2.2451
# Epoch: 0, Batch: 1150, Loss: 2.2455
# Epoch: 0, Batch: 1160, Loss: 2.2453
# Epoch: 0, Batch: 1170, Loss: 2.2452
# Epoch: 0, Batch: 1180, Loss: 2.2451
# Epoch: 0, Batch: 1190, Loss: 2.2452
# Epoch: 0, Batch: 1200, Loss: 2.2453
# Epoch: 0, Batch: 1210, Loss: 2.2458
# Epoch: 0, Batch: 1220, Loss: 2.2462
# Epoch: 0, Batch: 1230, Loss: 2.2462
# Epoch: 0, Batch: 1240, Loss: 2.2464

# Evaluating mAP...
```

[ ]:
```
# gpu_memory.txt

# |   2  NVIDIA GeForce RTX 2080 Ti     Off |   00000000:88:00.0 Off |         
↪        N/A |
# | 32%   56C    P2             217W /  250W |    10144MiB /  11264MiB |     81% 
↪     Default |
# |                                          |                        |         
↪        N/A |
```

[ ]:
```
# # inference.txt

# jupyter-st125457@puffer:~/rtml/a2_yolov4$ python detect.py --data data/coco.
↪yaml --weights checkpoints/yolov4.weights --img cocoimages/000000521540.jpg
# Loading YOLOv4 Model...
# Detected 2 objects!
# Class: spoon (44), Confidence: 0.8676, Box: [405.9, -22.0, 591.8, 220.0]
# Class: banana (46), Confidence: 0.9979, Box: [132.3, -53.0, 520.3, 520.9]

# Detection result saved to: /home/jupyter-st125457/rtml/a2_yolov4/results/
↪result_000000521540.jpg

# jupyter-st125457@puffer:~/rtml/a2_yolov4$ python run_yolov4.py
```

```
# Loading network.....
# Network successfully loaded
# <class 'numpy.ndarray'> (478, 640, 3)
# [ WARN:0@1.424] global loadsave.cpp:848 imwrite_ Unsupported depth image for␣
↪selected encoder is fallbacked to CV_8U.
# (608, 608, 3)
# (608, 608, 3)
# tensor([[608., 608., 608., 608.]])
# 000000581781.jpg     predicted in  0.438 seconds
# Objects Detected:    banana banana banana banana banana banana banana
# ----------------------------------------------------------
# Debug: c1=(285, 0), c2=(411, 114), type(c1)=<class 'tuple'>, type(c2)=<class␣
↪'tuple'>
# Debug: img.shape=(608, 608, 3), type(img)=<class 'numpy.ndarray'>
# Debug: c1=(127, 0), c2=(179, 108), type(c1)=<class 'tuple'>, type(c2)=<class␣
↪'tuple'>
# Debug: img.shape=(608, 608, 3), type(img)=<class 'numpy.ndarray'>
# Debug: c1=(135, 53), c2=(264, 179), type(c1)=<class 'tuple'>, type(c2)=<class␣
↪'tuple'>
# Debug: img.shape=(608, 608, 3), type(img)=<class 'numpy.ndarray'>
# Debug: c1=(343, 434), c2=(511, 608), type(c1)=<class 'tuple'>,␣
↪type(c2)=<class 'tuple'>
# Debug: img.shape=(608, 608, 3), type(img)=<class 'numpy.ndarray'>
# Debug: c1=(318, 250), c2=(549, 346), type(c1)=<class 'tuple'>,␣
↪type(c2)=<class 'tuple'>
# Debug: img.shape=(608, 608, 3), type(img)=<class 'numpy.ndarray'>
# Debug: c1=(188, 211), c2=(483, 334), type(c1)=<class 'tuple'>,␣
↪type(c2)=<class 'tuple'>
# Debug: img.shape=(608, 608, 3), type(img)=<class 'numpy.ndarray'>
# Debug: c1=(141, 450), c2=(318, 608), type(c1)=<class 'tuple'>,␣
↪type(c2)=<class 'tuple'>
# Debug: img.shape=(608, 608, 3), type(img)=<class 'numpy.ndarray'>
# SUMMARY
# ----------------------------------------------------------
# Task                       : Time Taken (in seconds)

# Reading addresses          : 0.000
# Loading batch              : 0.032
# Detection (2 images)       : 0.537
# Output Processing          : 0.000
# Drawing Boxes              : 0.006
# Average time_per_img       : 0.287

# jupyter-st125457@puffer:~/rtml/a2_yolov4$ python detect.py --img data/coco/
↪train2017/000000116031.jpg --weights checkpoints/yolov4.weights
# Loading YOLOv4 Model...
```

```
# Loading weights from checkpoints/yolov4.weights
# Loading image: /home/jupyter-st125457/rtml/a2_yolov4/data/coco/train2017/
↪000000116031.jpg
# Found 2 objects!
# Class: motorcycle (3), Confidence: 0.9774, Box: [-1.2, 5.8, 616.9, 598.8]
# Class: cat (15), Confidence: 0.9938, Box: [280.3, 193.2, 540.8, 475.3]

# Detection result saved to: /home/jupyter-st125457/rtml/a2_yolov4/results/
↪result_000000116031.jpg
# jupyter-st125457@puffer:~/rtml/a2_yolov4$ python detect.py --img data/coco/
↪train2017/000000233141.jpg --weights checkpoints/yolov4.weights
# Loading YOLOv4 Model...
# Loading weights from checkpoints/yolov4.weights
# Loading image: /home/jupyter-st125457/rtml/a2_yolov4/data/coco/train2017/
↪000000233141.jpg
# Found 2 objects!
# Class: person (0), Confidence: 0.9992, Box: [313.3, 10.4, 443.1, 218.8]
# Class: bench (13), Confidence: 0.9974, Box: [341.3, 20.7, 478.1, 197.9]

# Detection result saved to: /home/jupyter-st125457/rtml/a2_yolov4/results/
↪result_000000233141.jpg
# jupyter-st125457@puffer:~/rtml/a2_yolov4$ python detect.py --img data/coco/
↪train2017/000000523923.jpg --weights checkpoints/yolov4.weights
# Loading YOLOv4 Model...
# Loading weights from checkpoints/yolov4.weights
# Loading image: /home/jupyter-st125457/rtml/a2_yolov4/data/coco/train2017/
↪000000523923.jpg
# Found 9 objects!
# Class: person (0), Confidence: 0.9999, Box: [218.9, 97.3, 433.2, 489.6]
# Class: person (0), Confidence: 0.9996, Box: [262.3, 460.5, 339.1, 618.2]
# Class: person (0), Confidence: 0.9995, Box: [278.2, 312.2, 345.0, 507.8]
# Class: person (0), Confidence: 0.9996, Box: [313.8, 501.8, 351.9, 637.8]
# Class: person (0), Confidence: 0.9992, Box: [267.4, 395.8, 330.4, 558.8]
# Class: person (0), Confidence: 0.9998, Box: [327.7, 486.5, 368.2, 620.6]
# Class: person (0), Confidence: 0.9995, Box: [299.2, 364.6, 362.2, 535.1]
# Class: person (0), Confidence: 0.9897, Box: [292.9, 433.4, 370.1, 595.6]
# Class: skis (30), Confidence: 0.9985, Box: [459.8, 266.7, 579.4, 340.6]

# Detection result saved to: /home/jupyter-st125457/rtml/a2_yolov4/results/
↪result_000000523923.jpg
```

2 Conclusion

The project establishes a strong groundwork for YOLOv4 object detection using CIoU loss with only single epoch training. Its modular design ensures easy maintenance and potential enhancements. Our implementation successfully integrates all key components, including model architecture, data pipeline, loss computation, and evaluation metrics. The code is well-structured, adhering to best practices by organizing functionalities into separate modules. Additionally, robust error handling and logging mechanisms were incorporated throughout the pipeline to facilitate smooth training and inference, providing clear feedback on potential issues.

To validate the pipeline's functionality and ensure the model was training correctly, I conducted a single epoch of training as a preliminary test.

Trained models can be obtained via link