detection.docx

by Muhammad Nazir

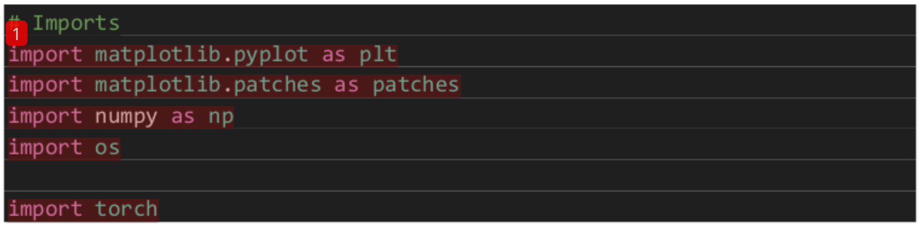
Submission date: 01-Aug-2024 11:18PM (UTC+1000)

Submission ID: 2423587036

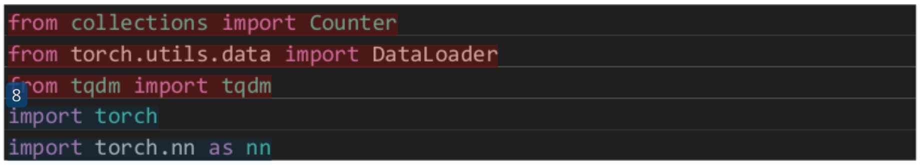
File name: detection.docx (72.8K)

Word count: 2811

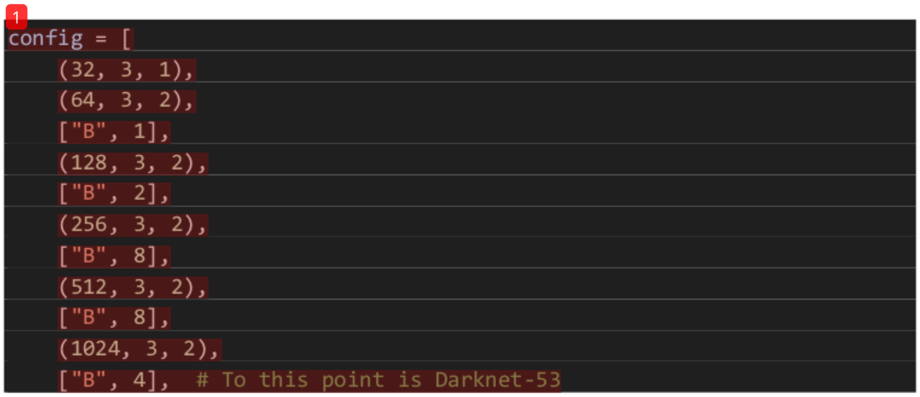
Character count: 19905



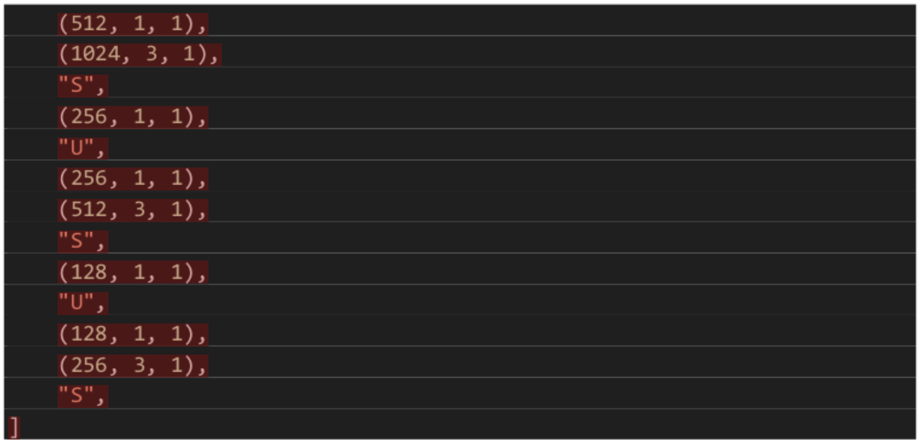
```
# Imports
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import numpy as np
import os

import torch
```



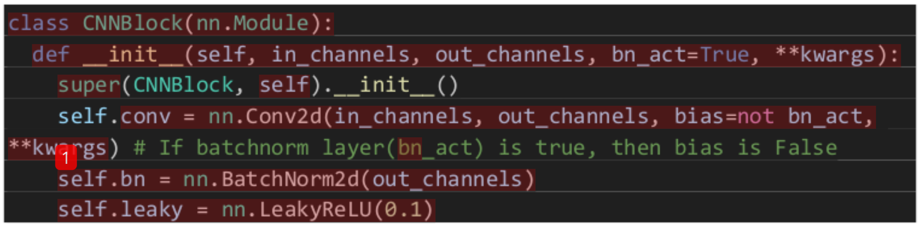
```
from collections import Counter
from torch.utils.data import DataLoader
from tqdm import tqdm
import torch
import torch.nn as nn
```



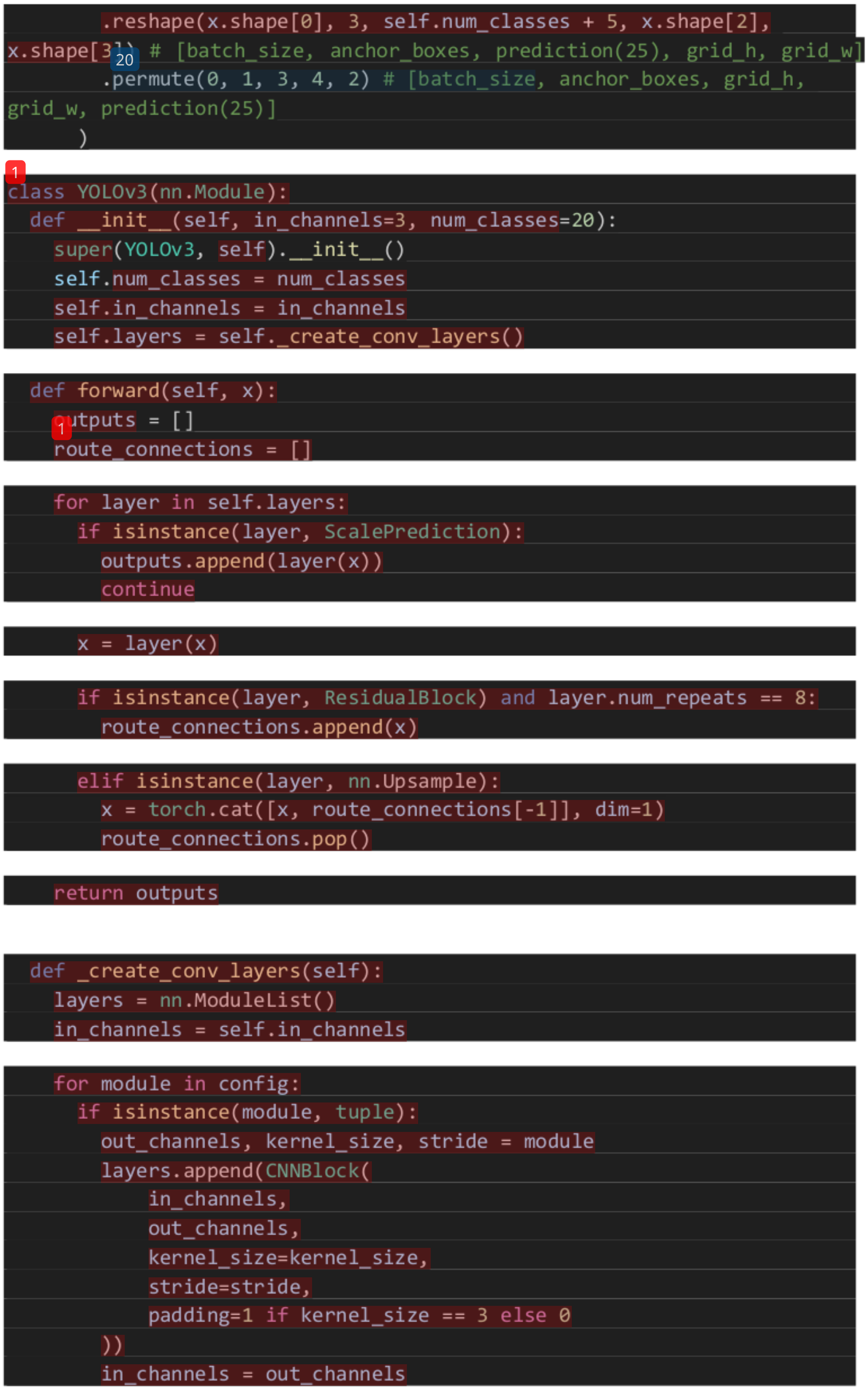
```
config = [
    (32, 3, 1),
    (64, 3, 2),
    ["B", 1],
    (128, 3, 2),
    ["B", 2],
    (256, 3, 2),
    ["B", 8],
    (512, 3, 2),
    ["B", 8],
    (1024, 3, 2),
    ["B", 4],  # To this point is Darknet-53
```



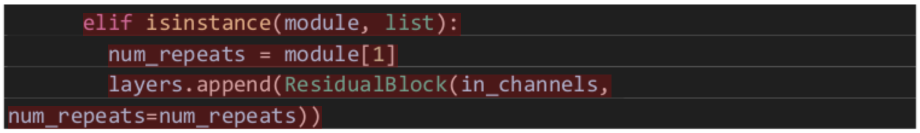
```
    (512, 1, 1),
    (1024, 3, 1),
    "S",
    (256, 1, 1),
    "U",
    (256, 1, 1),
    (512, 3, 1),
    "S",
    (128, 1, 1),
    "U",
    (128, 1, 1),
    (256, 3, 1),
    "S",
]
```



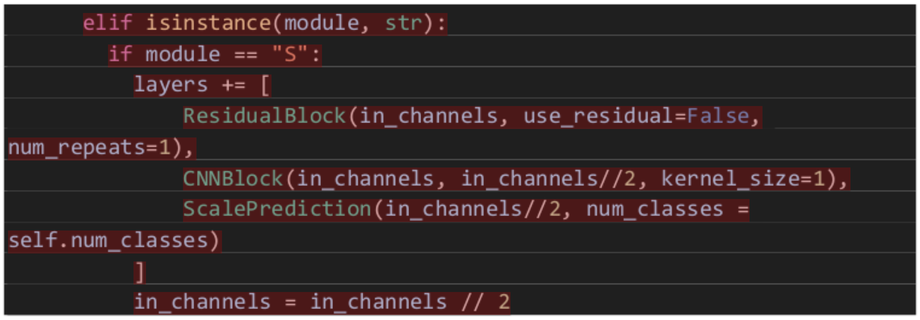
```
class CNNBlock(nn.Module):
  def __init__(self, in_channels, out_channels, bn_act=True, **kwargs):
    super(CNNBlock, self).__init__()
    self.conv = nn.Conv2d(in_channels, out_channels, bias=not bn_act,
**kwargs) # If batchnorm layer(bn_act) is true, then bias is False
    self.bn = nn.BatchNorm2d(out_channels)
    self.leaky = nn.LeakyReLU(0.1)
    self.use_bn_act = bn_act

  def forward(self, x):
    if self.use_bn_act:
      return self.leaky(self.bn(self.conv(x)))
    else:
      return self.conv(x)


class ResidualBlock(nn.Module):
  def __init__(self, channels, use_residual=True, num_repeats=1):
    super(ResidualBlock, self).__init__()
    self.layers = nn.ModuleList() # Like regular python list, but is
container for pytorch nn modules

    for repeat in range(num_repeats):
      self.layers += [
          nn.Sequential(
            CNNBlock(channels, channels//2, kernel_size=1),
            CNNBlock(channels//2, channels, kernel_size=3, padding=1)
          )
      ]

    self.use_residual = use_residual
    self.num_repeats = num_repeats

  def forward(self, x):
    for layer in self.layers:
      if self.use_residual:
        x = x + layer(x)
      else:
        x = layer(x)

    return x

class ScalePrediction(nn.Module):
  def __init__(self, in_channels, num_classes):
    super(ScalePrediction, self).__init__()
    self.pred = nn.Sequential(
        CNNBlock(in_channels, 2 * in_channels, kernel_size=3,
padding=1),
        CNNBlock(2 * in_channels, (num_classes + 5) * 3, bn_act=False,
kernel_size=1), # (num_classes + 5) * 3 --> (20+5) for each anchor box
which in total is 3
    )
    self.num_classes = num_classes

  def forward(self, x):
    return (
        self.pred(x)
```



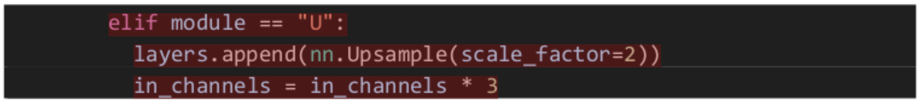
```python
        .reshape(x.shape[0], 3, self.num_classes + 5, x.shape[2],
x.shape[3]) # [batch_size, anchor_boxes, prediction(25), grid_h, grid_w]
        .permute(0, 1, 3, 4, 2) # [batch_size, anchor_boxes, grid_h,
grid_w, prediction(25)]
      )

class YOLOv3(nn.Module):
  def __init__(self, in_channels=3, num_classes=20):
    super(YOLOv3, self).__init__()
    self.num_classes = num_classes
    self.in_channels = in_channels
    self.layers = self._create_conv_layers()

  def forward(self, x):
    outputs = []
    route_connections = []

    for layer in self.layers:
      if isinstance(layer, ScalePrediction):
        outputs.append(layer(x))
        continue

      x = layer(x)

      if isinstance(layer, ResidualBlock) and layer.num_repeats == 8:
        route_connections.append(x)

      elif isinstance(layer, nn.Upsample):
        x = torch.cat([x, route_connections[-1]], dim=1)
        route_connections.pop()

    return outputs


  def _create_conv_layers(self):
    layers = nn.ModuleList()
    in_channels = self.in_channels

    for module in config:
      if isinstance(module, tuple):
        out_channels, kernel_size, stride = module
        layers.append(CNNBlock(
            in_channels,
            out_channels,
            kernel_size=kernel_size,
            stride=stride,
            padding=1 if kernel_size == 3 else 0
        ))
        in_channels = out_channels
```



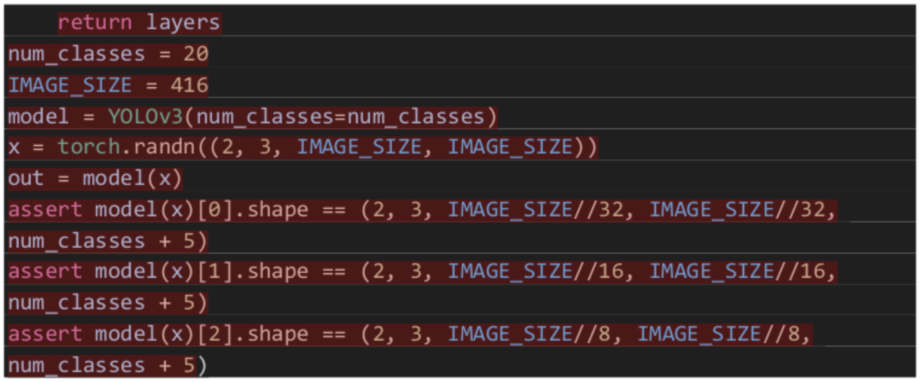
```
      elif isinstance(module, list):
        num_repeats = module[1]
        layers.append(ResidualBlock(in_channels,
num_repeats=num_repeats))
```



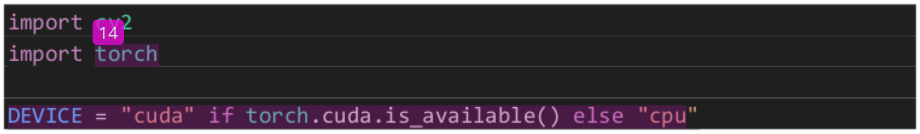
```
      elif isinstance(module, str):
        if module == "S":
          layers += [
              ResidualBlock(in_channels, use_residual=False,
num_repeats=1),
              CNNBlock(in_channels, in_channels//2, kernel_size=1),
              ScalePrediction(in_channels//2, num_classes =
self.num_classes)
          ]
          in_channels = in_channels // 2
```



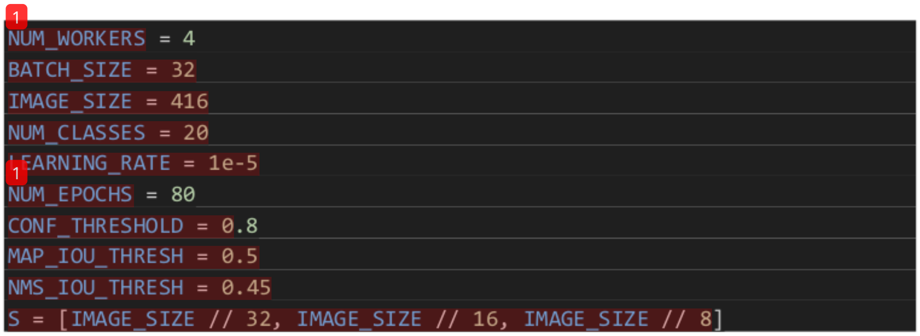
```
        elif module == "U":
          layers.append(nn.Upsample(scale_factor=2))
          in_channels = in_channels * 3
```



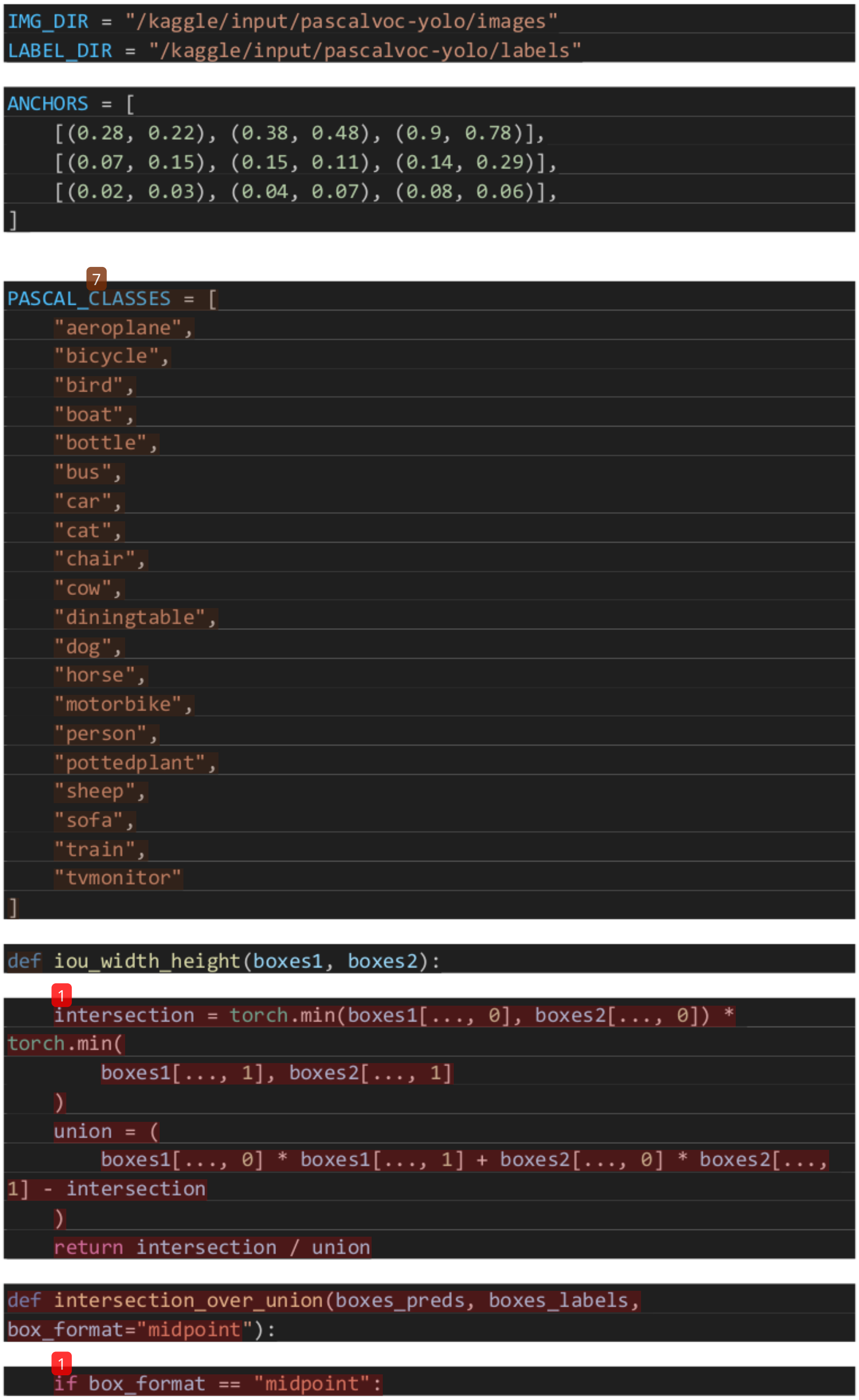
```
    return layers
num_classes = 20
IMAGE_SIZE = 416
model = YOLOv3(num_classes=num_classes)
x = torch.randn((2, 3, IMAGE_SIZE, IMAGE_SIZE))
out = model(x)
assert model(x)[0].shape == (2, 3, IMAGE_SIZE//32, IMAGE_SIZE//32,
num_classes + 5)
assert model(x)[1].shape == (2, 3, IMAGE_SIZE//16, IMAGE_SIZE//16,
num_classes + 5)
assert model(x)[2].shape == (2, 3, IMAGE_SIZE//8, IMAGE_SIZE//8,
num_classes + 5)
```



```
import cv2
import torch

DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
```



```
NUM_WORKERS = 4
BATCH_SIZE = 32
IMAGE_SIZE = 416
NUM_CLASSES = 20
LEARNING_RATE = 1e-5
NUM_EPOCHS = 80
CONF_THRESHOLD = 0.8
MAP_IOU_THRESH = 0.5
NMS_IOU_THRESH = 0.45
S = [IMAGE_SIZE // 32, IMAGE_SIZE // 16, IMAGE_SIZE // 8]
```



```
IMG_DIR = "/kaggle/input/pascalvoc-yolo/images"
LABEL_DIR = "/kaggle/input/pascalvoc-yolo/labels"
```

```
ANCHORS = [
    [(0.28, 0.22), (0.38, 0.48), (0.9, 0.78)],
    [(0.07, 0.15), (0.15, 0.11), (0.14, 0.29)],
    [(0.02, 0.03), (0.04, 0.07), (0.08, 0.06)],
]
```

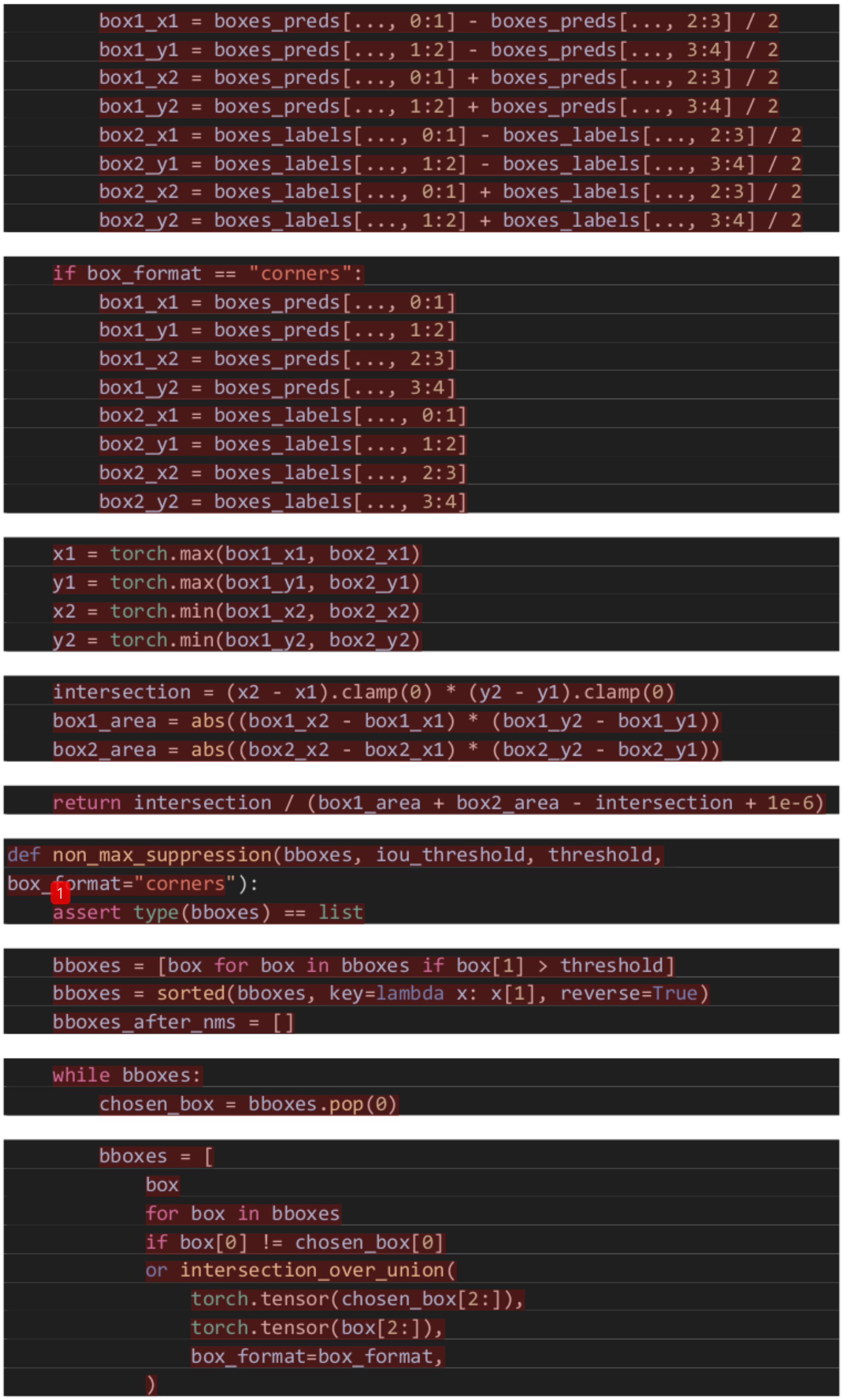
```
PASCAL_CLASSES = [
    "aeroplane",
    "bicycle",
    "bird",
    "boat",
    "bottle",
    "bus",
    "car",
    "cat",
    "chair",
    "cow",
    "diningtable",
    "dog",
    "horse",
    "motorbike",
    "person",
    "pottedplant",
    "sheep",
    "sofa",
    "train",
    "tvmonitor"
]
```

```
def iou_width_height(boxes1, boxes2):
```

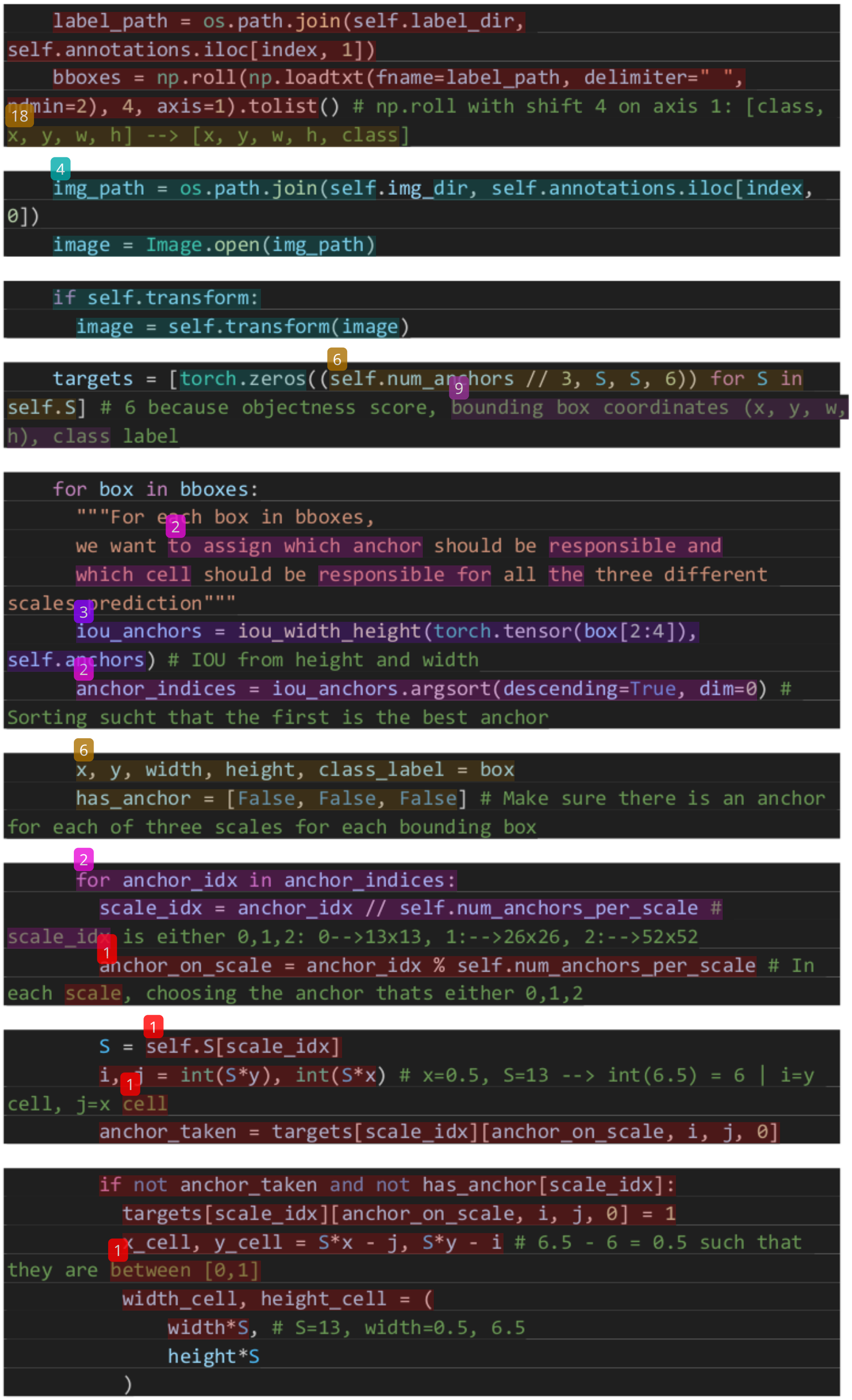
```
    intersection = torch.min(boxes1[..., 0], boxes2[..., 0]) *
torch.min(
        boxes1[..., 1], boxes2[..., 1]
    )
    union = (
        boxes1[..., 0] * boxes1[..., 1] + boxes2[..., 0] * boxes2[...,
1] - intersection
    )
    return intersection / union
```

```
def intersection_over_union(boxes_preds, boxes_labels,
box_format="midpoint"):
```

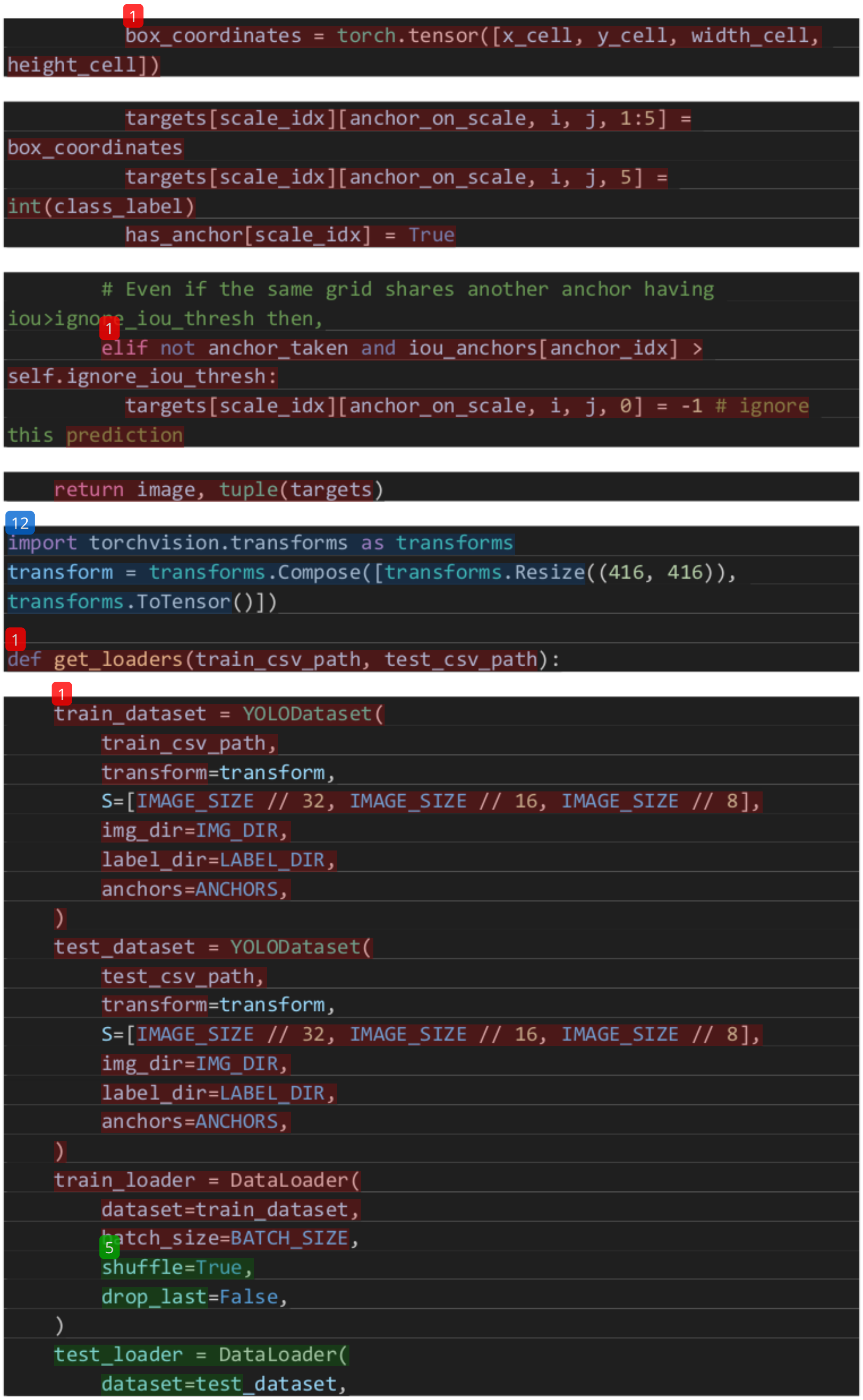
```
    if box_format == "midpoint":
```



```
        box1_x1 = boxes_preds[..., 0:1] - boxes_preds[..., 2:3] / 2
        box1_y1 = boxes_preds[..., 1:2] - boxes_preds[..., 3:4] / 2
        box1_x2 = boxes_preds[..., 0:1] + boxes_preds[..., 2:3] / 2
        box1_y2 = boxes_preds[..., 1:2] + boxes_preds[..., 3:4] / 2
        box2_x1 = boxes_labels[..., 0:1] - boxes_labels[..., 2:3] / 2
        box2_y1 = boxes_labels[..., 1:2] - boxes_labels[..., 3:4] / 2
        box2_x2 = boxes_labels[..., 0:1] + boxes_labels[..., 2:3] / 2
        box2_y2 = boxes_labels[..., 1:2] + boxes_labels[..., 3:4] / 2

    if box_format == "corners":
        box1_x1 = boxes_preds[..., 0:1]
        box1_y1 = boxes_preds[..., 1:2]
        box1_x2 = boxes_preds[..., 2:3]
        box1_y2 = boxes_preds[..., 3:4]
        box2_x1 = boxes_labels[..., 0:1]
        box2_y1 = boxes_labels[..., 1:2]
        box2_x2 = boxes_labels[..., 2:3]
        box2_y2 = boxes_labels[..., 3:4]

    x1 = torch.max(box1_x1, box2_x1)
    y1 = torch.max(box1_y1, box2_y1)
    x2 = torch.min(box1_x2, box2_x2)
    y2 = torch.min(box1_y2, box2_y2)

    intersection = (x2 - x1).clamp(0) * (y2 - y1).clamp(0)
    box1_area = abs((box1_x2 - box1_x1) * (box1_y2 - box1_y1))
    box2_area = abs((box2_x2 - box2_x1) * (box2_y2 - box2_y1))

    return intersection / (box1_area + box2_area - intersection + 1e-6)

def non_max_suppression(bboxes, iou_threshold, threshold,
box_format="corners"):
    assert type(bboxes) == list

    bboxes = [box for box in bboxes if box[1] > threshold]
    bboxes = sorted(bboxes, key=lambda x: x[1], reverse=True)
    bboxes_after_nms = []

    while bboxes:
        chosen_box = bboxes.pop(0)

        bboxes = [
            box
            for box in bboxes
            if box[0] != chosen_box[0]
            or intersection_over_union(
                torch.tensor(chosen_box[2:]),
                torch.tensor(box[2:]),
                box_format=box_format,
            )
            < iou_threshold
        ]

        bboxes_after_nms.append(chosen_box)

    return bboxes_after_nms
import numpy as np
import os
import pandas as pd
import torch

from torch.utils.data import Dataset, DataLoader
from PIL import Image, ImageFile

# allows PIL to load images even if they are truncated or incomplete
ImageFile.LOAD_TRUNCATED_IMAGES = True

class YOLODataset(Dataset):
  def __init__(self, csv_file, img_dir, label_dir, anchors,
               image_size=416, S=[13,26,52], C=20, transform=None):
    self.annotations = pd.read_csv(csv_file)
    self.img_dir = img_dir
    self.label_dir = label_dir
    self.transform = transform
    self.S = S

    # Suppose, anchors[0] = [a,b,c], anchors[1] = [d,e,f], anchors[2] =
[g,h,i] : Each set of anchors for each scale
    # List addition gives shape 3x3
    # Anchors per scale suggests that there are three different aspect
ratios for each anchor position.
    self.anchors = torch.tensor(anchors[0] + anchors[1] + anchors[2]) #
For all 3 scales
    self.num_anchors = self.anchors.shape[0]
    self.num_anchors_per_scale = self.num_anchors // 3

    self.C = C

    # If a cell has obj. then one anchor is responsible for outputting
it,
    # one that's responsible is the one that has highest iou with
ground truth box
    # but, there might be cases where there are several boxes in the
same cell
    self.ignore_iou_thresh = 0.5

  def __len__(self):
    return len(self.annotations)

  def __getitem__(self, index):
```



```
    label_path = os.path.join(self.label_dir,
self.annotations.iloc[index, 1])
    bboxes = np.roll(np.loadtxt(fname=label_path, delimiter=" ",
ndmin=2), 4, axis=1).tolist() # np.roll with shift 4 on axis 1: [class,
x, y, w, h] --> [x, y, w, h, class]

    img_path = os.path.join(self.img_dir, self.annotations.iloc[index,
0])
    image = Image.open(img_path)

    if self.transform:
      image = self.transform(image)

    targets = [torch.zeros((self.num_anchors // 3, S, S, 6)) for S in
self.S] # 6 because objectness score, bounding box coordinates (x, y, w,
h), class label

    for box in bboxes:
      """For each box in bboxes,
      we want to assign which anchor should be responsible and
      which cell should be responsible for all the three different
scales prediction"""
      iou_anchors = iou_width_height(torch.tensor(box[2:4]),
self.anchors) # IOU from height and width
      anchor_indices = iou_anchors.argsort(descending=True, dim=0) #
Sorting sucht that the first is the best anchor

      x, y, width, height, class_label = box
      has_anchor = [False, False, False] # Make sure there is an anchor
for each of three scales for each bounding box

      for anchor_idx in anchor_indices:
        scale_idx = anchor_idx // self.num_anchors_per_scale #
scale_idx is either 0,1,2: 0-->13x13, 1:-->26x26, 2:-->52x52
        anchor_on_scale = anchor_idx % self.num_anchors_per_scale # In
each scale, choosing the anchor thats either 0,1,2

        S = self.S[scale_idx]
        i, j = int(S*y), int(S*x) # x=0.5, S=13 --> int(6.5) = 6 | i=y
cell, j=x cell
        anchor_taken = targets[scale_idx][anchor_on_scale, i, j, 0]

        if not anchor_taken and not has_anchor[scale_idx]:
          targets[scale_idx][anchor_on_scale, i, j, 0] = 1
          x_cell, y_cell = S*x - j, S*y - i # 6.5 - 6 = 0.5 such that
they are between [0,1]
          width_cell, height_cell = (
              width*S, # S=13, width=0.5, 6.5
              height*S
          )
```



```
          box_coordinates = torch.tensor([x_cell, y_cell, width_cell,
height_cell])
```

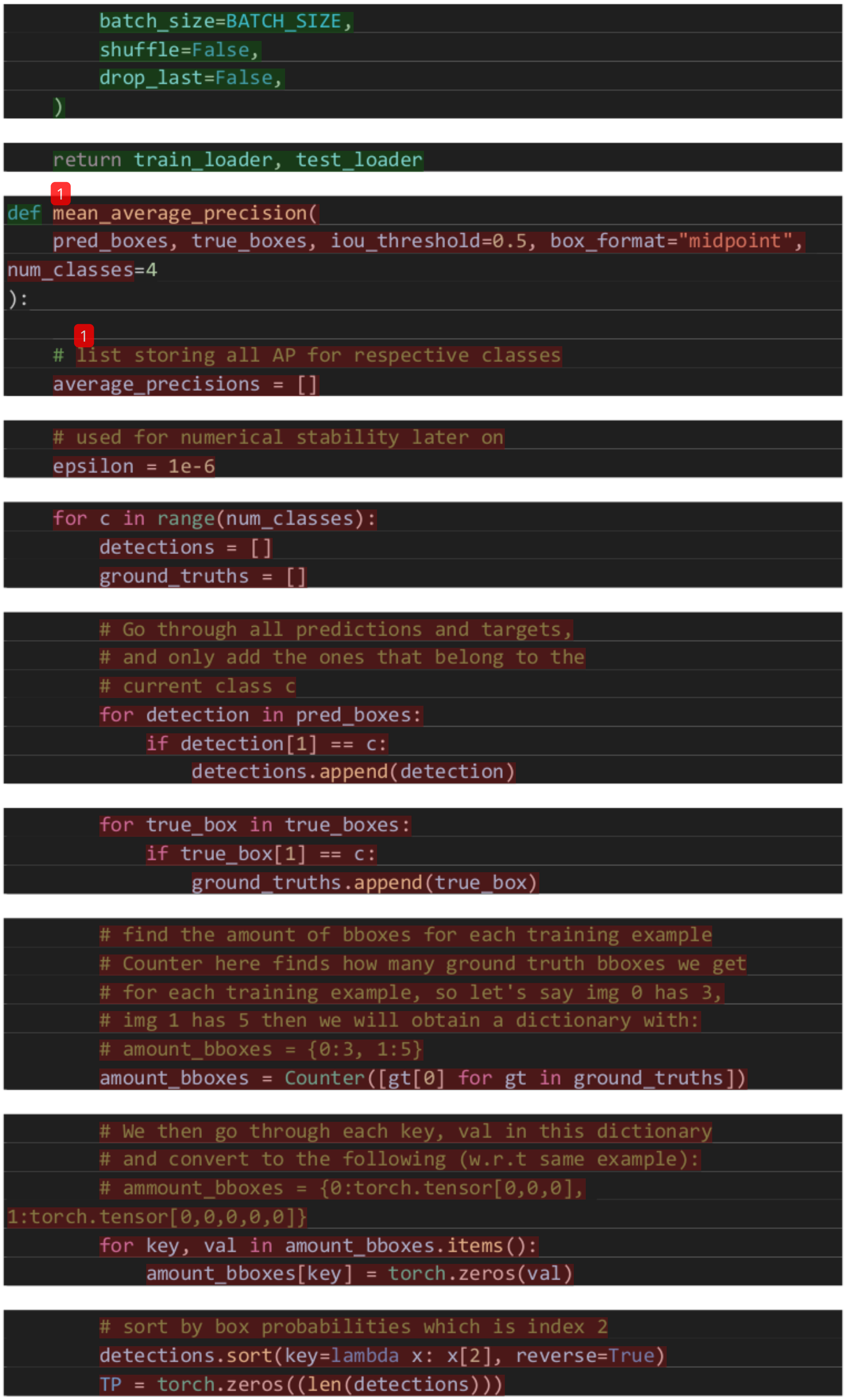
```
          targets[scale_idx][anchor_on_scale, i, j, 1:5] =
box_coordinates
          targets[scale_idx][anchor_on_scale, i, j, 5] =
int(class_label)
          has_anchor[scale_idx] = True
```

```
        # Even if the same grid shares another anchor having
iou>ignore_iou_thresh then,
        elif not anchor_taken and iou_anchors[anchor_idx] >
self.ignore_iou_thresh:
          targets[scale_idx][anchor_on_scale, i, j, 0] = -1 # ignore
this prediction
```

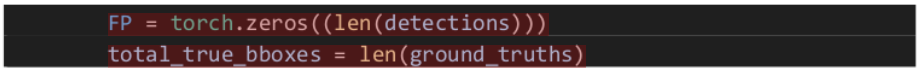
```
    return image, tuple(targets)
```

```
import torchvision.transforms as transforms
transform = transforms.Compose([transforms.Resize((416, 416)),
transforms.ToTensor()])

def get_loaders(train_csv_path, test_csv_path):
```

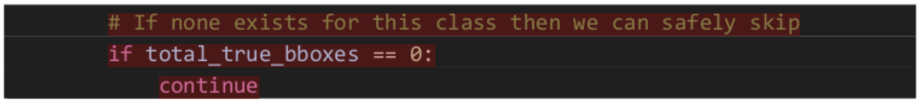
```
    train_dataset = YOLODataset(
        train_csv_path,
        transform=transform,
        S=[IMAGE_SIZE // 32, IMAGE_SIZE // 16, IMAGE_SIZE // 8],
        img_dir=IMG_DIR,
        label_dir=LABEL_DIR,
        anchors=ANCHORS,
    )
    test_dataset = YOLODataset(
        test_csv_path,
        transform=transform,
        S=[IMAGE_SIZE // 32, IMAGE_SIZE // 16, IMAGE_SIZE // 8],
        img_dir=IMG_DIR,
        label_dir=LABEL_DIR,
        anchors=ANCHORS,
    )
    train_loader = DataLoader(
        dataset=train_dataset,
        batch_size=BATCH_SIZE,
        shuffle=True,
        drop_last=False,
    )
    test_loader = DataLoader(
        dataset=test_dataset,
```



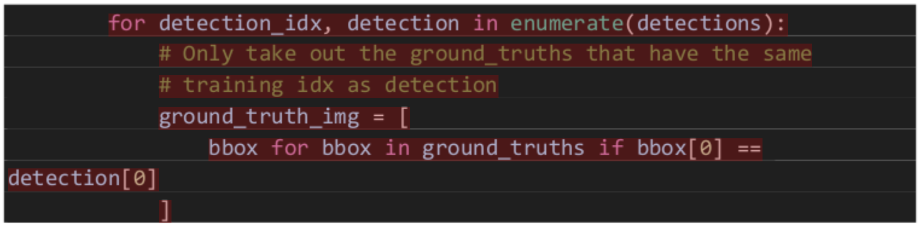
```
        batch_size=BATCH_SIZE,
        shuffle=False,
        drop_last=False,
    )

    return train_loader, test_loader

def mean_average_precision(
    pred_boxes, true_boxes, iou_threshold=0.5, box_format="midpoint",
num_classes=4
):

    # list storing all AP for respective classes
    average_precisions = []

    # used for numerical stability later on
    epsilon = 1e-6

    for c in range(num_classes):
        detections = []
        ground_truths = []

        # Go through all predictions and targets,
        # and only add the ones that belong to the
        # current class c
        for detection in pred_boxes:
            if detection[1] == c:
                detections.append(detection)

        for true_box in true_boxes:
            if true_box[1] == c:
                ground_truths.append(true_box)

        # find the amount of bboxes for each training example
        # Counter here finds how many ground truth bboxes we get
        # for each training example, so let's say img 0 has 3,
        # img 1 has 5 then we will obtain a dictionary with:
        # amount_bboxes = {0:3, 1:5}
        amount_bboxes = Counter([gt[0] for gt in ground_truths])

        # We then go through each key, val in this dictionary
        # and convert to the following (w.r.t same example):
        # ammount_bboxes = {0:torch.tensor[0,0,0],
1:torch.tensor[0,0,0,0,0]}
        for key, val in amount_bboxes.items():
            amount_bboxes[key] = torch.zeros(val)

        # sort by box probabilities which is index 2
        detections.sort(key=lambda x: x[2], reverse=True)
        TP = torch.zeros((len(detections)))
```



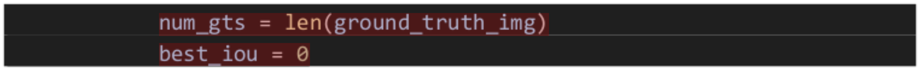
```
        FP = torch.zeros((len(detections)))
        total_true_bboxes = len(ground_truths)
```



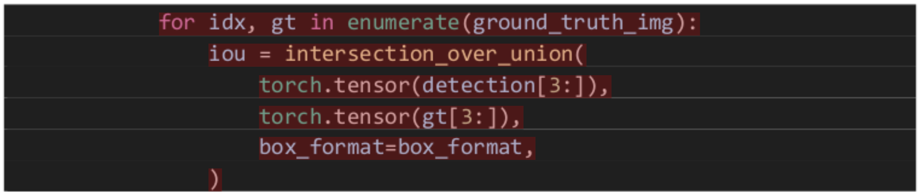
```
        # If none exists for this class then we can safely skip
        if total_true_bboxes == 0:
            continue
```



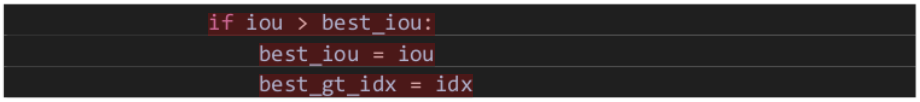
```
        for detection_idx, detection in enumerate(detections):
            # Only take out the ground_truths that have the same
            # training idx as detection
            ground_truth_img = [
                bbox for bbox in ground_truths if bbox[0] ==
detection[0]
            ]
```



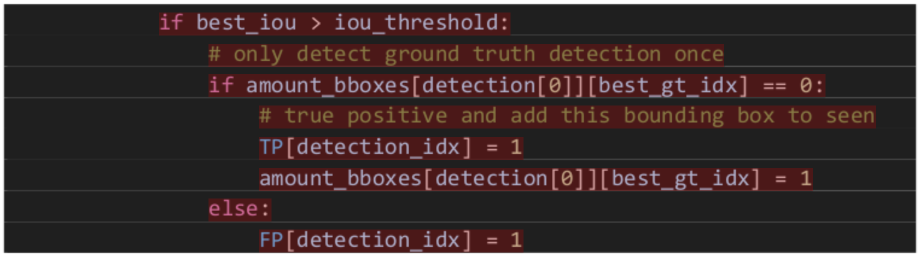
```
            num_gts = len(ground_truth_img)
            best_iou = 0
```



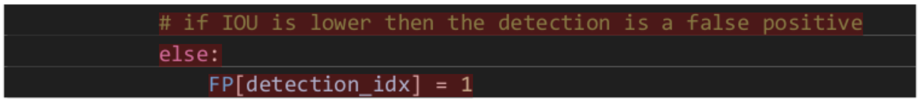
```
            for idx, gt in enumerate(ground_truth_img):
                iou = intersection_over_union(
                    torch.tensor(detection[3:]),
                    torch.tensor(gt[3:]),
                    box_format=box_format,
                )
```



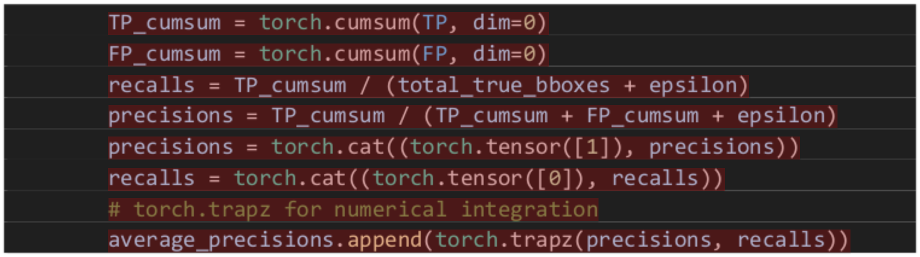
```
                if iou > best_iou:
                    best_iou = iou
                    best_gt_idx = idx
```



```
            if best_iou > iou_threshold:
                # only detect ground truth detection once
                if amount_bboxes[detection[0]][best_gt_idx] == 0:
                    # true positive and add this bounding box to seen
                    TP[detection_idx] = 1
                    amount_bboxes[detection[0]][best_gt_idx] = 1
                else:
                    FP[detection_idx] = 1
```



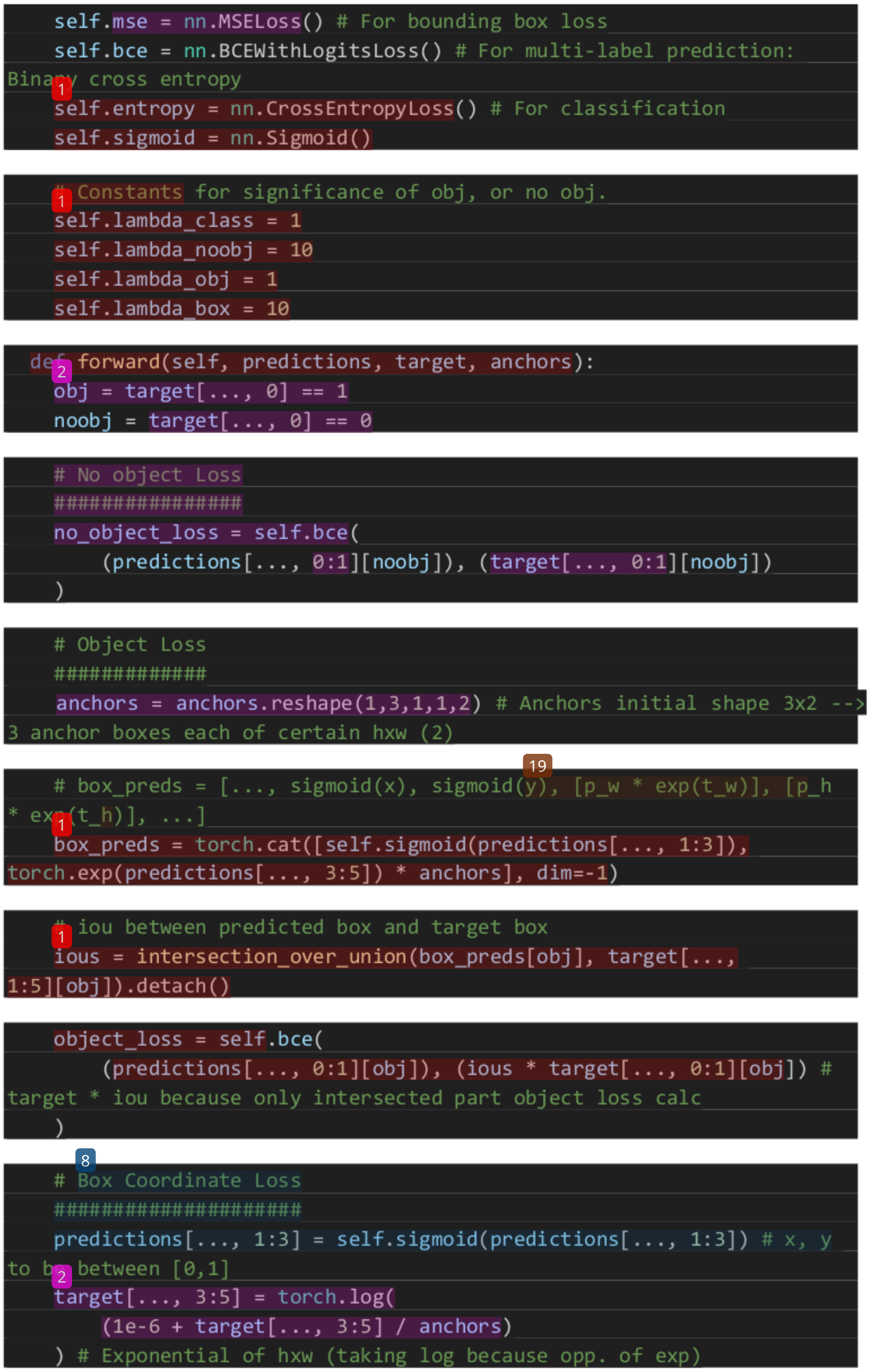
```
            # if IOU is lower then the detection is a false positive
            else:
                FP[detection_idx] = 1
```



```
        TP_cumsum = torch.cumsum(TP, dim=0)
        FP_cumsum = torch.cumsum(FP, dim=0)
        recalls = TP_cumsum / (total_true_bboxes + epsilon)
        precisions = TP_cumsum / (TP_cumsum + FP_cumsum + epsilon)
        precisions = torch.cat((torch.tensor([1]), precisions))
        recalls = torch.cat((torch.tensor([0]), recalls))
        # torch.trapz for numerical integration
        average_precisions.append(torch.trapz(precisions, recalls))
```

```
    return sum(average_precisions) / len(average_precisions)

def get_evaluation_bboxes(
    loader,
    model,
    iou_threshold,
    anchors,
    threshold,
    box_format="midpoint",
    device="cuda" if torch.cuda.is_available() else "cpu",
):
    # make sure model is in eval before get bboxes
    model.eval()
    train_idx = 0
    all_pred_boxes = []
    all_true_boxes = []
    for batch_idx, (x, labels) in enumerate(loader):
        x = x.float().to(device)

        with torch.no_grad():
            predictions = model(x)

        batch_size = x.shape[0]
        bboxes = [[] for _ in range(batch_size)]
        for i in range(3):
            S = predictions[i].shape[2] # grid cell size for each
predictions
            anchor = torch.tensor([*anchors[i]]).to(device) * S #
anchor for each grid, prediction type
            boxes_scale_i = cells_to_bboxes( # get bboxes for each
image in the batch
                predictions[i], anchor, S=S, is_preds=True
            )
            for idx, (box) in enumerate(boxes_scale_i): # for each
image, append the bbox to corr. bboxes[idx]
                bboxes[idx] += box

        # we just want one bbox for each label, not one for each scale
        true_bboxes = cells_to_bboxes(
            labels[2], anchor, S=S, is_preds=False
        )

        for idx in range(batch_size):
            nms_boxes = non_max_suppression(
                bboxes[idx],
                iou_threshold=iou_threshold,
                threshold=threshold,
                box_format=box_format,
```

```python
            )

            for nms_box in nms_boxes:
                all_pred_boxes.append([train_idx] + nms_box)

            for box in true_bboxes[idx]:
                if box[1] > threshold:
                    all_true_boxes.append([train_idx] + box)

            train_idx += 1

    model.train()
    return all_pred_boxes, all_true_boxes

def cells_to_bboxes(predictions, anchors, S, is_preds=True):

    BATCH_SIZE = predictions.shape[0]
    num_anchors = len(anchors)
    box_predictions = predictions[..., 1:5]
    if is_preds:
        anchors = anchors.reshape(1, len(anchors), 1, 1, 2)
        box_predictions[..., 0:2] = torch.sigmoid(box_predictions[...,
0:2])
        box_predictions[..., 2:] = torch.exp(box_predictions[..., 2:])
* anchors
        scores = torch.sigmoid(predictions[..., 0:1])
        best_class = torch.argmax(predictions[..., 5:], dim=-
1).unsqueeze(-1)
    else:
        scores = predictions[..., 0:1]
        best_class = predictions[..., 5:6]

    cell_indices = (
        torch.arange(S)
        .repeat(predictions.shape[0], 3, S, 1)
        .unsqueeze(-1)
        .to(predictions.device)
    )
    x = 1 / S * (box_predictions[..., 0:1] + cell_indices)
    y = 1 / S * (box_predictions[..., 1:2] + cell_indices.permute(0, 1,
3, 2, 4))
    w_h = 1 / S * box_predictions[..., 2:4]
    converted_bboxes = torch.cat((best_class, scores, x, y, w_h), dim=-
1).reshape(BATCH_SIZE, num_anchors * S * S, 6)
    return converted_bboxes.tolist()

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



```
  self.mse = nn.MSELoss() # For bounding box loss
  self.bce = nn.BCEWithLogitsLoss() # For multi-label prediction:
Binary cross entropy
  self.entropy = nn.CrossEntropyLoss() # For classification
  self.sigmoid = nn.Sigmoid()

  # Constants for significance of obj, or no obj.
  self.lambda_class = 1
  self.lambda_noobj = 10
  self.lambda_obj = 1
  self.lambda_box = 10

 def forward(self, predictions, target, anchors):
  obj = target[..., 0] == 1
  noobj = target[..., 0] == 0

  # No object Loss
  ################
  no_object_loss = self.bce(
    (predictions[..., 0:1][noobj]), (target[..., 0:1][noobj])
  )

  # Object Loss
  #############
  anchors = anchors.reshape(1,3,1,1,2) # Anchors initial shape 3x2 -->
3 anchor boxes each of certain hxw (2)

  # box_preds = [..., sigmoid(x), sigmoid(y), [p_w * exp(t_w)], [p_h
* exp(t_h)], ...]
  box_preds = torch.cat([self.sigmoid(predictions[..., 1:3]),
torch.exp(predictions[..., 3:5]) * anchors], dim=-1)

  # iou between predicted box and target box
  ious = intersection_over_union(box_preds[obj], target[...,
1:5][obj]).detach()

  object_loss = self.bce(
    (predictions[..., 0:1][obj]), (ious * target[..., 0:1][obj]) #
target * iou because only intersected part object loss calc
  )

  # Box Coordinate Loss
  #####################
  predictions[..., 1:3] = self.sigmoid(predictions[..., 1:3]) # x, y
to be between [0,1]
  target[..., 3:5] = torch.log(
    (1e-6 + target[..., 3:5] / anchors)
  ) # Exponential of hxw (taking log because opp. of exp)
```

```
    box_loss = self.mse(predictions[..., 1:5][obj], target[...,
1:5][obj])
```

```
    # Class Loss
    ############
    class_loss = self.entropy(
        (predictions[..., 5:][obj]), (target[..., 5][obj].long())
    )
```

```
    return(
        self.lambda_box * box_loss
        + self.lambda_obj * object_loss
        + self.lambda_noobj * no_object_loss
        + self.lambda_class * class_loss
    )
```

```
def plot_image(image, boxes):
    """Plots predicted bounding boxes on the image"""
    cmap = plt.get_cmap("tab20b")
    class_labels = PASCAL_CLASSES
    colors = [cmap(i) for i in np.linspace(0, 1, len(class_labels))]
    im = np.array(image)
    height, width, _ = im.shape
```

```
    # Create figure and axes
    fig, ax = plt.subplots(1)
    # Display the image
    ax.imshow(im)
```

```
    # box[0] is x midpoint, box[2] is width
    # box[1] is y midpoint, box[3] is height
```

```
    # Create a Rectangle patch
    for box in boxes:
        assert len(box) == 6, "box should contain class pred,
confidence, x, y, width, height"
        class_pred = box[0]
        box = box[2:]
        upper_left_x = box[0] - box[2] / 2
        upper_left_y = box[1] - box[3] / 2
        rect = patches.Rectangle(
            (upper_left_x * width, upper_left_y * height),
            box[2] * width,
            box[3] * height,
            linewidth=2,
            edgecolor=colors[int(class_pred)],
            facecolor="none",
        )
        # Add the patch to the Axes
        ax.add_patch(rect)
        plt.text(
            upper_left_x * width,
            upper_left_y * height,
            s=class_labels[int(class_pred)],
            color="white",
            verticalalignment="top",
            bbox={"color": colors[int(class_pred)], "pad": 0},
        )
```

```
    plt.show()

    # Save test loader to a file
torch.save(test_loader, '/kaggle/working/test_loader.pth')
```

```
import torch.optim as optim

from tqdm import tqdm
import time
```

```
history_loss = [] # To plot the epoch vs. loss
```

```
for epoch in tqdm(range(NUM_EPOCHS), desc="Epochs"):
  model.train()
```

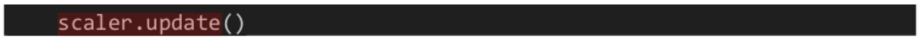
```
  losses = []
```

```
  start_time = time.time() # Start time of the epoch
```

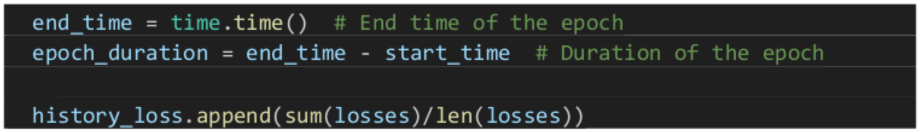
```
  for batch_idx, (x,y) in enumerate(train_loader):
    x = x.to(DEVICE)
    y0, y1, y2 = (y[0].to(DEVICE),
                  y[1].to(DEVICE),
                  y[2].to(DEVICE))
```

```
    # context manager is used in PyTorch to automatically handle mixed-precision computations on CUDA-enabled GPUs
    with torch.cuda.amp.autocast():
      out = model(x)
      loss = (
          loss_fn(out[0], y0, scaled_anchors[0])
          + loss_fn(out[1], y1, scaled_anchors[1])
          + loss_fn(out[2], y2, scaled_anchors[2])
      )
```

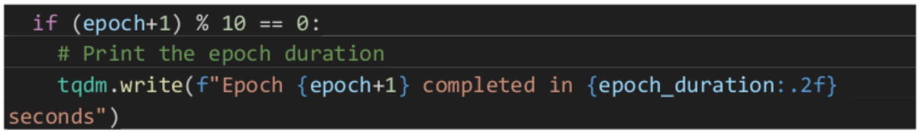
```
    losses.append(loss.item())

    optimizer.zero_grad()
    scaler.scale(loss).backward()
    scaler.step(optimizer)
```



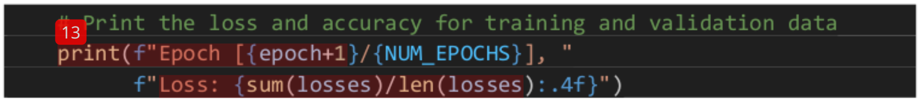
```
    scaler.update()
```



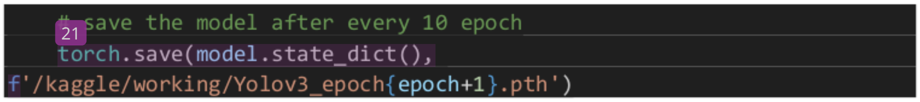
```
  end_time = time.time()  # End time of the epoch
  epoch_duration = end_time - start_time  # Duration of the epoch

  history_loss.append(sum(losses)/len(losses))
```



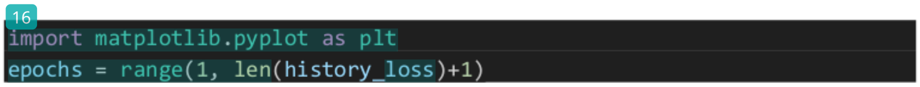
```
  if (epoch+1) % 10 == 0:
    # Print the epoch duration
    tqdm.write(f"Epoch {epoch+1} completed in {epoch_duration:.2f}
seconds")
```



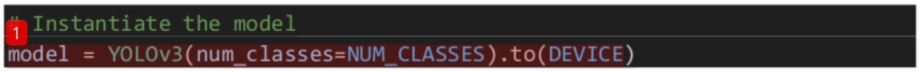
```
    # Print the loss and accuracy for training and validation data
    print(f"Epoch [{epoch+1}/{NUM_EPOCHS}], "
          f"Loss: {sum(losses)/len(losses):.4f}")
```



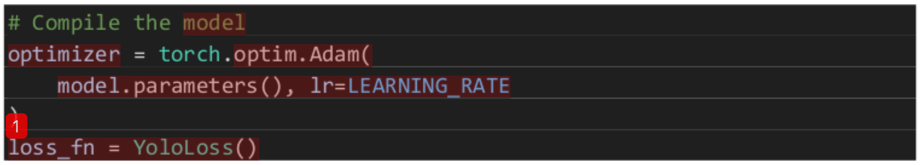
```
    # Save the model after every 10 epoch
    torch.save(model.state_dict(),
f'/kaggle/working/Yolov3_epoch{epoch+1}.pth')
```



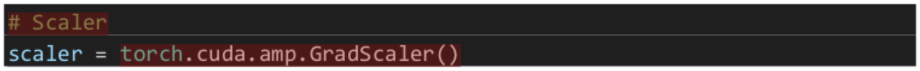
```
import matplotlib.pyplot as plt
epochs = range(1, len(history_loss)+1)
```



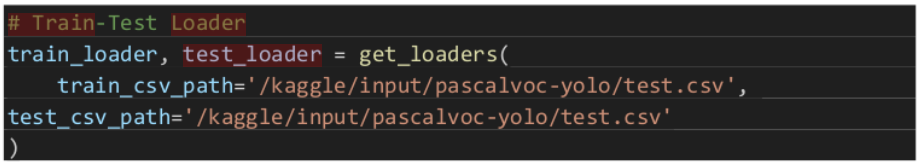
```
# Instantiate the model
model = YOLOv3(num_classes=NUM_CLASSES).to(DEVICE)
```



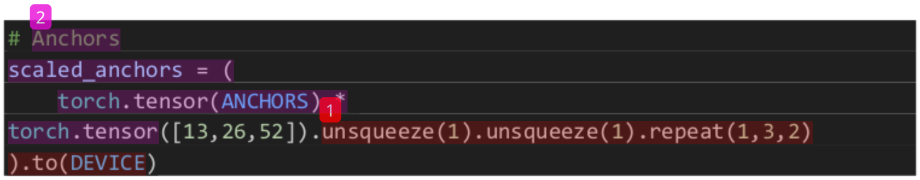
```
# Compile the model
optimizer = torch.optim.Adam(
    model.parameters(), lr=LEARNING_RATE
)
loss_fn = YoloLoss()
```



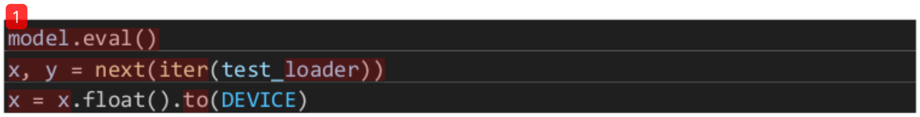
```
# Scaler
scaler = torch.cuda.amp.GradScaler()
```



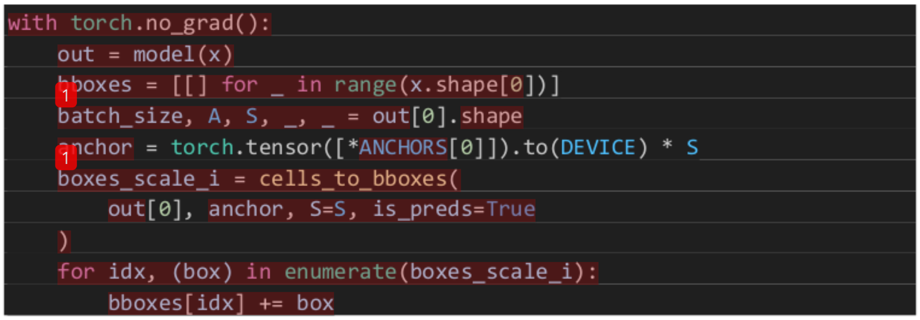
```
# Train-Test Loader
train_loader, test_loader = get_loaders(
    train_csv_path='/kaggle/input/pascalvoc-yolo/test.csv',
test_csv_path='/kaggle/input/pascalvoc-yolo/test.csv'
)
```



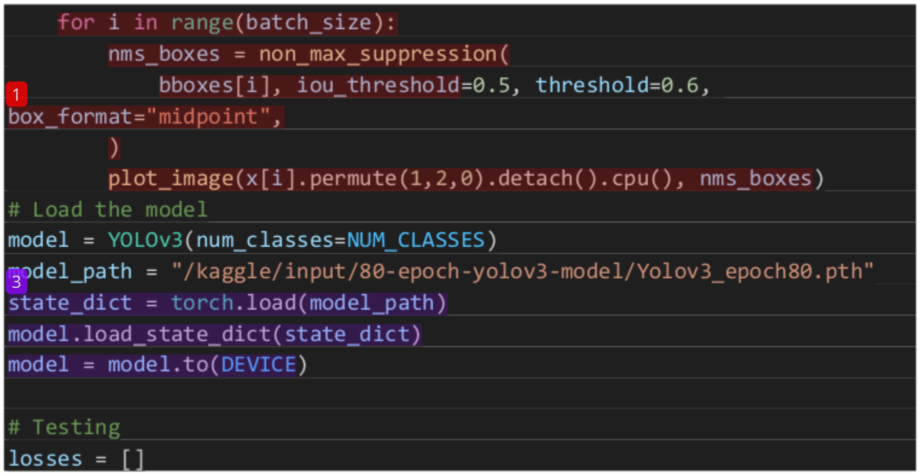
```
# Anchors
scaled_anchors = (
    torch.tensor(ANCHORS) *
torch.tensor([13,26,52]).unsqueeze(1).unsqueeze(1).repeat(1,3,2)
).to(DEVICE)
```



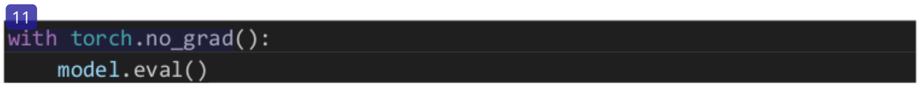
```
model.eval()
x, y = next(iter(test_loader))
x = x.float().to(DEVICE)
```



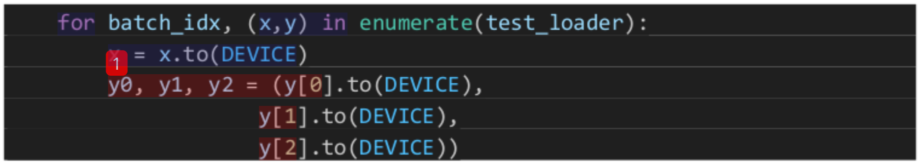
```python
with torch.no_grad():
    out = model(x)
    bboxes = [[] for _ in range(x.shape[0])]
    batch_size, A, S, _, _ = out[0].shape
    anchor = torch.tensor([*ANCHORS[0]]).to(DEVICE) * S
    boxes_scale_i = cells_to_bboxes(
        out[0], anchor, S=S, is_preds=True
    )
    for idx, (box) in enumerate(boxes_scale_i):
        bboxes[idx] += box
```



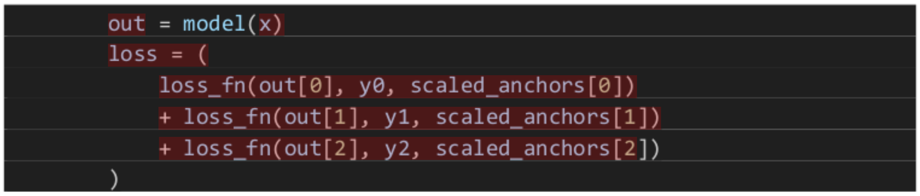
```python
    for i in range(batch_size):
        nms_boxes = non_max_suppression(
            bboxes[i], iou_threshold=0.5, threshold=0.6,
box_format="midpoint",
        )
        plot_image(x[i].permute(1,2,0).detach().cpu(), nms_boxes)
# Load the model
model = YOLOv3(num_classes=NUM_CLASSES)
model_path = "/kaggle/input/80-epoch-yolov3-model/Yolov3_epoch80.pth"
state_dict = torch.load(model_path)
model.load_state_dict(state_dict)
model = model.to(DEVICE)

# Testing
losses = []
```



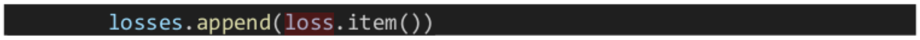
```python
with torch.no_grad():
    model.eval()
```



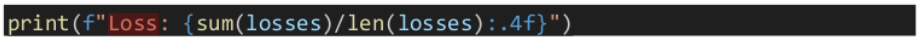
```python
    for batch_idx, (x,y) in enumerate(test_loader):
        x = x.to(DEVICE)
        y0, y1, y2 = (y[0].to(DEVICE),
                      y[1].to(DEVICE),
                      y[2].to(DEVICE))
```



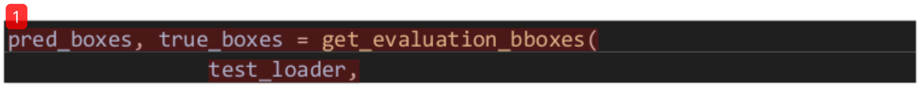
```python
        out = model(x)
        loss = (
            loss_fn(out[0], y0, scaled_anchors[0])
            + loss_fn(out[1], y1, scaled_anchors[1])
            + loss_fn(out[2], y2, scaled_anchors[2])
        )
```



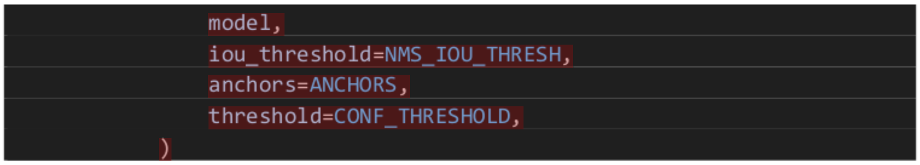
```python
        losses.append(loss.item())
```



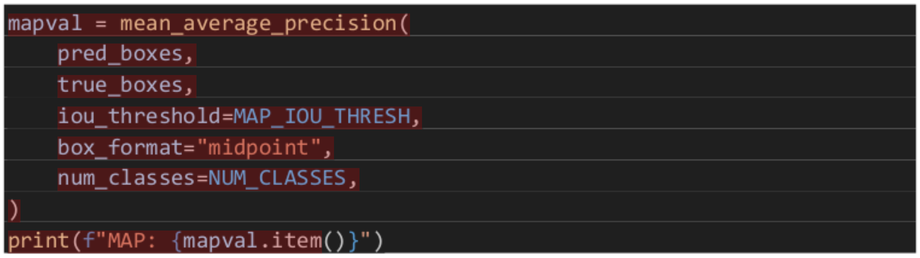
```python
print(f"Loss: {sum(losses)/len(losses):.4f}")
```



```python
pred_boxes, true_boxes = get_evaluation_bboxes(
                test_loader,
```



```
                model,
                iou_threshold=NMS_IOU_THRESH,
                anchors=ANCHORS,
                threshold=CONF_THRESHOLD,
            )
```



```
mapval = mean_average_precision(
    pred_boxes,
    true_boxes,
    iou_threshold=MAP_IOU_THRESH,
    box_format="midpoint",
    num_classes=NUM_CLASSES,
)
print(f"MAP: {mapval.item()}")
```

detection.docx

ORIGINALITY REPORT

81%	81%	13%	28%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	huggingface.co Internet Source	71%
2	origin.geeksforgeeks.org Internet Source	3%
3	discuss.pytorch.org Internet Source	1%
4	Submitted to University of East London Student Paper	1%
5	Submitted to University of Sydney Student Paper	1%
6	u-n-joe.tistory.com Internet Source	1%
7	git.trustie.net Internet Source	1%
8	hdl.handle.net Internet Source	1%
9	mdpi-res.com Internet Source	<1%

10	Submitted to City University Student Paper	<1%
11	Submitted to University of North Texas Student Paper	<1%
12	ctkim.tistory.com Internet Source	<1%
13	Eklas Hossain. "Machine Learning Crash Course for Engineers", Springer Science and Business Media LLC, 2024 Publication	<1%
14	deep-learning-study.tistory.com Internet Source	<1%
15	kypseli.ouc.ac.cy Internet Source	<1%
16	openi.pcl.ac.cn Internet Source	<1%
17	raschka-research-group.github.io Internet Source	<1%
18	velog.io Internet Source	<1%
19	Shancheng Fang, Hongtao Xie, Zhineng Chen, Shiai Zhu, Xiaoyan Gu, Xingyu Gao. "Detecting Uyghur text in complex background images with convolutional neural network", Multimedia Tools and Applications, 2017	<1%

	Publication	
20	www.codetd.com Internet Source	<1 %
21	Adnan, Tamim. "Classification and Pixel-Level Segmentation of Asphalt Pavement Cracks Using Convolutional Neural Networks", The University of North Carolina at Charlotte, 2023 Publication	<1 %

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	On		