```
# Drive setup
from google.colab import drive
drive.mount('/content/drive')
```

```
    Drive already mounted at /content/drive; to attempt to forcibly remount, call
```

```
import torch
import torch.nn as nn
import torch.optim as optim
from tqdm.auto import tqdm
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import sampler
from torchvision import datasets
import torch.nn.functional as F
import torchvision.datasets as dset
import torchvision.transforms as T
from google.colab.patches import cv2_imshow
from torchvision.transforms import ToTensor

import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow
import cv2
import pickle
import os
import pandas as pd
import random
import numpy as np
from torchvision.io import ImageReadMode, read_image
from torch.utils.data import DataLoader
import matplotlib.image as mpimg
import cv2
import pickle
import warnings
import copy
import pickle
from google.colab.patches import cv2_imshow
import random
from sklearn.metrics import average_precision_score
import math
import torchvision
warnings.filterwarnings('ignore')
```

Data Preparation

```
with open('/content/drive/MyDrive/SSD detection/images_2.pkl', 'rb') as f:
    images = pickle.load(f)
with open('/content/drive/MyDrive/SSD detection/labels_2.pkl', 'rb') as f:
    labels = pickle.load(f)
labels = [[label] for label in labels]
with open('/content/drive/MyDrive/SSD detection/coco_person_fire_hydrant_image_3.p
    images_rest = pickle.load(f)
    # images_rest = []
for idx in range(len(images_rest)):
  images_rest[idx] = np.array(images_rest[idx])
with open('/content/drive/MyDrive/SSD detection/coco_person_fire_hydrant_class_lab
    labels_rest = pickle.load(f)
    # labels_rest = []
with open('/content/drive/MyDrive/SSD detection/coco_person_fire_hydrant_annotatio
    annotations = pickle.load(f)

random.Random(4).shuffle(images)
random.Random(4).shuffle(labels)
random.Random(5).shuffle(images_rest)
random.Random(5).shuffle(labels_rest)
random.Random(5).shuffle(annotations)
```

```
print(len(images))
print(len(labels))
print(len(images_rest))
print(len(labels_rest))
print(len(annotations))
# print(len(images_door))
# print(len(annotations_door))
```

```
459
459
438
438
438
```

```
print(labels[:5])
print(labels_rest[:5])
print(annotations[:5])
# print(annotations_door[:5])
```

```
[[[137, 91, 226, 217]], [[96, 109, 123, 142]], [[88, 176, 163, 281]], [[67, 1
[['fire hydrant', 'person', 'person'], ['person', 'person', 'fire hydrant'],
[[[104, 123, 237, 313], [32, 19, 156, 283], [184, 34, 295, 276]], [[33, 47, 1
```

Processing images in COCO - Making sure that very small humans are not taken into consideration in the model (harm the results a lot).

```
images_coco = []
labels_coco = []
annotations_coco = []
val = 0
for img in images_rest:
    cnt = 0
    label_here = []
    annotation_here = []
    z = 0
    if len(annotations[val]) > 4:
      val = val + 1
      continue
    for label in annotations[val]:
      # try:
        # Checking width and person class
        height, width = img.shape[:2]
        if ((label[2]-label[0])*(label[3]-label[1]))/(width*height) < 0.005:
            continue
      # except:
        # print(label)
        annotation_here.append(label)
        label_here.append(labels_rest[val][z])
        cnt += 1
        z += 1
    val += 1
    if cnt > 0:
        images_coco.append(img)
        labels_coco.append(label_here)
        annotations_coco.append(annotation_here)

# print(len(images_coco))
```

Removing weird images in the dataset (Black & White/malformed).

```
def remove_wrong_images(images, labels, annotations = None):
    images_new = []
    labels_new = []
    annotations_new = []
    cnt = 0
    for cnt in range(len(images)):
        # Checking malformed images
        if len(images[cnt].shape) < 3 or images[cnt].shape[2] != 3:
            continue
        images_new.append(images[cnt])
        labels_new.append(labels[cnt])
        if annotations is not None:
            annotations_new.append(annotations[cnt])
    return images_new, labels_new, annotations_new

images_coco, annotations_coco, labels_coco = remove_wrong_images(images_coco, anno
# images_door, annotations_door, _ = remove_wrong_images(images_door, annotations_
images, labels, _ = remove_wrong_images(images, labels)
```

```
def image_augmentation_flip_horizontal(images):
  original_images = images
  new_images = []
  for idx in range(len(new_images)):
    image = cv2.flip(new_images[idx], 1)#*255
    new_images.append(image)
    # cv2_imshow(image)
  original_images.extend(new_images)
  return original_images
```

```
import copy
def image_augmentation_translation(images_list, labels_list, annotations_list):
  original_images_list = copy.deepcopy(images_list)
  original_labels_list = copy.deepcopy(labels_list)
  original_annotations_list = copy.deepcopy(annotations_list)
  new_images_list = []
  new_labels_list = []
  new_annotations_list = []
  for img, labels, annotations in zip(images_list, labels_list, annotations_list):
    # annotations = [annotations]
    height, width = img.shape[:2]
    translation_factor = 0.2
    max_horizontal = translation_factor*width
    max_vertical = translation_factor*height
    horizontal_shift = int(random.uniform(-max_horizontal, max_horizontal))
    vertical_shift = int(random.uniform(-max_vertical, max_vertical))
    # M[0][2] = how much right
    # M[1][2] = how much down
    M = np.float32([[1, 0, horizontal_shift], [0, 1, vertical_shift]])
    new_img = cv2.warpAffine(img, M, (width, height))
    # filling blank spaces in image
    if horizontal_shift >= 0:
      new_img[:,:horizontal_shift] = np.random.rand(320, horizontal_shift, 3)
    else:
     new_img[:,horizontal_shift:] = np.random.rand(320, -horizontal_shift, 3)
    if vertical_shift >= 0:
      new_img[:vertical_shift,:] = np.random.rand(vertical_shift, 320,3)
    else:
      new_img[vertical_shift:,:] = np.random.rand(-vertical_shift, 320,3)

    # new_images.append(new_img)
    temp_label = []
    temp_annotations = []
    for label, box in zip(labels, annotations):
      x1 = box[0]
      y1 = box[1]
      x2 = box[2]
      y2 = box[3]
      x1 = x1 + horizontal_shift
      y1 = y1 + vertical_shift
      x2 = x2 + horizontal_shift
      y2 = y2 + vertical_shift
      if x1 >= 0 and x1 < width and y1 >= 0 and y1 < height and x2 >= 0 and x2 < w
        temp_label.append(label) # No cutting
        temp_annotations.append([x1, y1, x2, y2]) # No cutting just translation
      elif x1 >= width or y1 >= height or x2 < 0 or y2 < 0:
        pass # whole out
      else: # for cases in which there is partial image cut [if >=30% is image has
        # find new coordinates of bbox
        initial_area = (x2-x1)*(y2-y1)
        if x1 < 0:
          x1 = 0
        if y1 < 0:
          y1 = 0
        if x2 >= width:
          x2 = width - 1
        if y2 >= height:
          y2 = height - 1
        final_area = (x2-x1)*(y2-y1)
        # check if more than 50 % has gone out
        if final_area >= 0.7*initial_area:
          # accept modified
          temp_annotations.append([x1, y1, x2, y2])
          temp_label.append(label)
        else:
          pass
        # do something
    if len(temp_annotations) > 0:
      new_images_list.append(new_img)
      new_labels_list.append(temp_label)
      new_annotations_list.append(temp_annotations)

    # print(horizontal_shift, vertical_shift)
    # cv2_imshow(translated)
  result_images_list = []
  result_labels_list = []
  result_annotations_list = []
  for img1, img2 in zip(original_images_list, new_images_list):
    result_images_list.append(img1)
    result_images_list.append(img2)
  for label1, label2 in zip(original_labels_list, new_labels_list):
    result_labels_list.append(label1)
    result_labels_list.append(label2)
  for annotation1, annotation2 in zip(original_annotations_list, new_annotations_l
    result_annotations_list.append(annotation1)
    result_annotations_list.append(annotation2)

  return result_images_list, result_labels_list, result_annotations_list

index1 = random.randint(0, len(images_coco))
index2 = random.randint(0, len(images_coco))
index3 = random.randint(0, len(images_coco))
dummy_img_list = [images_coco[index1], images_coco[index2], images_coco[index3]]
dummy_class_list = [labels_coco[index1], labels_coco[index2], labels_coco[index3]]
dummy_annotations_list = [annotations_coco[index1], annotations_coco[index2], anno

# dummy_img_list, dummy_class_list, dummy_annotations_list = image_augmentation_tra
# images, dummy_class_list, labels = image_augmentation_translation(images, [[1]]*
# images_coco, labels_coco, annotations_coco = image_augmentation_translation(image

import copy
def img_list_viewer(images_list, labels_list, annotations_list):
  for img, labels, annotations in zip(images_list, labels_list, annotations_list):
    temp_img = copy.deepcopy(img)
    # print(labels, annotations)
    for label, annotation in zip(labels, annotations):
      # print(label, annotation)
      color = None
      if label == "doll":
        color = (1, 0, 0)
      elif label == "fire hydrant":
        color = (0, 1 ,0)
      elif label == "person":
        color = (0, 0 ,1)
      # print(annotations)
      temp_img = cv2.rectangle(temp_img, (annotation[0], annotation[1]), (annotati
    temp_img = temp_img*255
    temp_img = np.array(temp_img, dtype=np.uint8)
    temp_img = cv2.cvtColor(temp_img, cv2.COLOR_RGB2BGR)
    cv2_imshow(temp_img)
  return

dummy_img, dummy_label, dummy_annotation = image_augmentation_translation(images[:
img_list_viewer(dummy_img, dummy_label, dummy_annotation)
```

```
# counting no of samples per class
print("No. of doll samples : ", len(images))
count_person = 0
count_fire_hydrant = 0
flat_labels_coco = [item for sublist in labels_coco for item in sublist]
for class_name in flat_labels_coco:
  if class_name == "person":
    count_person = count_person + 1
  elif class_name == "fire hydrant":
    count_fire_hydrant = count_fire_hydrant + 1
# print(flat_labels_coco[:5])
print("No of person samples : ", count_person)
print("No of fire hydrant samples : ", count_fire_hydrant)
```

```
No. of doll samples :  459
No of person samples :  481
No of fire hydrant samples :  305
```

```
dummy_img, dummy_label, dummy_annotation = image_augmentation_translation(images_r
img_list_viewer(dummy_img, dummy_label, dummy_annotation)
```

```
images, dummy_list, labels = image_augmentation_translation(images, [["doll"]]*len
images_coco, labels_coco, annotations_coco= image_augmentation_translation(images_
```

```
images, dummy_list, labels = image_augmentation_translation(images, [["doll"]]*len
images_coco, labels_coco, annotations_coco= image_augmentation_translation(images_
```

```
# counting no of samples per class
print("No. of doll samples : ", len(images))
```

```
count_person = 0
count_fire_hydrant = 0
flat_labels_coco = [item for sublist in labels_coco for item in sublist]
for class_name in flat_labels_coco:
  if class_name == "person":
    count_person = count_person + 1
  elif class_name == "fire hydrant":
    count_fire_hydrant = count_fire_hydrant + 1
# print(flat_labels_coco[:5])
print("No of person samples : ", count_person)
print("No of fire hydrant samples : ", count_fire_hydrant)
```

```
No. of doll samples :  1608
No of person samples :  1655
No of fire hydrant samples :  1073
```

Data Augmentation

The following image transform classes were defined to deal with bounding boxes in transformations.

1. Random Crop (resized to full)
2. Perspective Transformation

```
# For reference, see PyTorch's implementation of T.RandomResizedCrop
class RandomResizedCropWithBox(T.RandomResizedCrop):
    def __init__(self, *args, **kwargs):
        super(RandomResizedCropWithBox, self).__init__(*args, **kwargs)

    def forward(self, img_data):
        img = img_data[0]
        boxes = img_data[1]
        classes = img_data[2]
        i, j, h, w = self.get_params(img, self.scale, self.ratio)
        num_boxes = len(boxes)
        new_labels = []
        new_classes = []

        # Creating labels for each bounding box
        for val in range(num_boxes):
            xmin, ymin, xmax, ymax = boxes[val]
            # Checking if it lies inside the cropped image
            if xmax <= j or xmin >= j+w or ymax <= i or ymin >= i+h:
                continue
            x1 = (max(xmin, j)-j)*320/w
            x2 = (min(xmax, j+w)-j)*320/w
            y1 = (max(ymin, i)-i)*320/h
            y2 = (min(ymax, i+h)-i)*320/h
            for value in (x1, x2, y1, y2):
                if value < 0:
                    value = 0
                if value >= 320:
                    value = 319
            new_labels.append([x1, y1, x2, y2])
            new_classes.append(classes[val])
        new_labels = torch.from_numpy(np.array(new_labels)).int()

        # Returns resized image, labels (box co-ordinates) and classes
        return [torchvision.transforms.functional.resized_crop(img, i, j, h, w, se
```

The custom dataset class which outputs objects according to `idx` - Different ranges give objects from different arrays above.

`collate_fn` is used to get the correct format of values from the dataloader (to handle the dictionaries correctly).

```
class DollDataset(Dataset):

    # All arrays and values that are part of the Dataset class
    def __init__(self, images, labels, images_coco, labels_coco, annotations, xsiz
        self.images = images
        self.labels = labels
        self.images_rest = images_coco
        self.labels_rest = labels_coco
        self.annotations = annotations
        # self.images_doors = images_doors
        # self.labels_doors = labels_doors
        self.xsize = xsize
        self.ysize = ysize
        self.perspective_prob = perspective_prob

    # Combining 3 arrays
    def __len__(self):
        return len(self.images)+len(self.images_rest)#+len(self.images_doors)

    # Crucial function, returns (non-)transformed image with box
    def __getitem__(self, idx):

        # Normalize with calculated mean and std dev
        trans = T.Compose([T.ToTensor(), T.Resize((self.xsize,self.ysize)), T.Norma
        # trans = T.Compose([T.ToTensor(), T.Resize((self.xsize,self.ysize))])

        # Dealing with individual arrays
        z = len(self.images)
        z2 = len(self.images_rest)
        # print(z, z+z2)
        if idx < z:
            class_list = [1]
        elif idx < z+z2:
            class_list = self.labels_rest[idx-z]
            for i in range(len(class_list)):
                if class_list[i] == 'person':
                    class_list[i] = 3
                if class_list[i] == 'fire hydrant':
                    class_list[i] = 2
        # else:
```

```
        # else:
        #     class_list = [3]*len(self.labels_doors[idx-z-z2])
        try:
          if idx < z:
              img = self.images[idx]
          elif idx < z+z2:
              img = self.images_rest[idx-z]
        except:
          print("from img ", idx)
        # else:
        #     img = self.images_doors[idx-z-z2]
        try:
          if idx < z:
              label = np.array(self.labels[idx])
          elif idx < z+z2:
              label = np.array(self.annotations[idx-z])
          else:
            print("from label ", idx)
        except:
          print("from label ", idx)
        # else:
        #     label = np.array(self.labels_doors[idx-z-z2])

        # Label resizing
        label = torch.from_numpy(label)
        y_size, x_size = img.shape[:2]
        label[:,1] = label[:,1]*320/y_size
        label[:,3] = label[:,3]*320/y_size
        label[:,0] = label[:,0]*320/x_size
        label[:,2] = label[:,2]*320/x_size
        label = label.int()
        # print(img)
        img = trans(img)
        return (img, label, class_list)

# Necessary to form a dataloader
def collate_fn(data):
    dics = []
    for x in range(len(data)):
        dic = {'image': data[x][0], 'bbox': data[x][1], 'label': torch.tensor(data
        dics.append(dic)
    return dics

print(len(images)*4//5, len(images_coco)*4//5)

a = len(images)*4//5
b = len(images_coco)*4//5



# train dataset
random.Random(6).shuffle(images[:a])
random.Random(6).shuffle(labels[:a])
random.Random(6).shuffle(images_coco[:b])
random.Random(6).shuffle(labels_coco[:b])
random.Random(6).shuffle(annotations_coco[:b])
train_set = DollDataset(images[:a], labels[:a], images_coco[:b], labels_coco[:b], a
train_set = DollDataset(images[:a], labels[:a], images_coco[:b], labels_coco[:b], a

print(len(images[a:]), len(images_coco[b:]))
# test dataset
random.Random(2).shuffle(images[a:])
random.Random(2).shuffle(labels[a:])
random.Random(2).shuffle(images_coco[b:])
random.Random(2).shuffle(labels_coco[b:])
random.Random(2).shuffle(annotations_coco[b:])
val_set = DollDataset(images[a:], labels[a:], images_coco[b:], labels_coco[b:], ann

# print(train_set.__len__())
# print(val_set.__len__())

# Training and Validation
# train_set, _ = torch.utils.data.random_split(doll_set1, [doll_set1.__len__()*1, (
# val_set, _ = torch.utils.data.random_split(doll_set2, [doll_set2.__len__()*1, 0]
train_loader = DataLoader(train_set, batch_size = 16, shuffle = True, collate_fn =
val_loader = DataLoader(val_set, batch_size = 16, shuffle = True, collate_fn = col
```

```
1286 1156
322 290
```

```
# print(doll_set[0][0])
```

```
# print(doll_set[7][0])
```

```
print(len(train_set), len(val_set))
```

```
2442 612
```

```
# train_set[2430][1]
```

```
# Testing if the dataloader works
for dic in tqdm(train_loader):
    break
```

Testing the working of the dataloader using the `show` function for torch tensors.

```
# a =
```

```
# import torchvision.transforms.functional as F
# import random

# random.seed(0)
# torch.manual_seed(0)
# np.random.seed(0)
# # Custom function to display images
# def show(imgs):
#     if not isinstance(imgs, list):
#         imgs = [imgs]
#     fix, axs = plt.subplots(ncols=len(imgs), squeeze=False)
#     for i, img in enumerate(imgs):
#         img = img.detach()
#         img = F.to_pil_image(img)
#         axs[0, i].imshow(np.asarray(img))
#         axs[0, i].set(xticklabels=[], yticklabels=[], xticks=[], yticks=[])

# from torchvision.transforms.functional import convert_image_dtype
# from torchvision.utils import draw_bounding_boxes
# plt.rcParams["savefig.bbox"] = 'tight'
# # T.Normalize([70.0594, 62.4050, 58.8377], [78.0825, 72.5514, 73.4684]
# # Showing one batch of images coming from the data loader
# for dic in tqdm(val_loader):
#     for x in range(len(dic)):
#         z = dic[x]['image']
#         z[0] = (z[0]*0.2736+0.4662)*255
#         z[1] = (z[1]*0.2650+0.4279)*255
#         z[2] = (z[2]*0.2774+0.3946)*255
#         # print(type(z))
#         # print(z)
#         bbox = dic[x]['bbox']
#         # print(bbox)
#         boxes = []
#         boxes.append(torch.tensor([0,0,0,0]))
#         for al in range(len(bbox)):
#             boxes.append(bbox[al])
#         img=draw_bounding_boxes(z.type(torch.uint8), boxes=torch.vstack(boxes),
#         # img =
#         show(img)
#         break
```

Useful functions

```
iou_threshold = 0.5 # for mAP and Confusion matrix
nms_iou_threshold = 0.3
```

```
# Calculates IOU
def iou(box1, box2):
    # box1 = list(map(lambda x: int(x), box1))
    # box2 = list(map(lambda x: int(x), box2))
    a1 = (box1[2]-box1[0])*(box1[3]-box1[1])
    a2 = (box2[2]-box2[0])*(box2[3]-box2[1])
    inter = max(0, min(box1[2], box2[2]) - max(box1[0], box2[0])) * max(0, min(box
    return inter/(a1 + a2 - inter)
```

```
def mean_average_precision(output, target, iou_threshold = iou_threshold, starting
final_map = 0
final_map_1 = 0
final_map_2 = 0
final_map_3 = 0
# ap_per_class = [0, 0, 0, 0]
# count = [0, 0, 0, 0]
for k in range(len(output)):
  confidence = [[],[],[],[]]
  pos_neg = [[],[],[],[]]
  if output[k]['labels'].size()[0] == 0:
      if target[k]['labels'].size()[0] == 0:
          final_map += 1
          continue
      else:
          final_map += 0
          continue
  for i in range(output[k]['labels'].size()[0]):
      confidence[output[k]['labels'][i]].append(float(output[k]['scores'][i]))
      pos_neg[output[k]['labels'][i]].append(False)
      for j in range(target[k]['labels'].size()[0]):
          if target[k]['labels'][j] != output[k]['labels'][i]:
              continue
          if iou(target[k]['boxes'][j].tolist(), output[k]['boxes'].int()[i].t
              pos_neg[output[k]['labels'][i]][-1] = True
              break
  # thresholds = np.arange(start=0.2, stop=0.7, step=0.05)

  sum_ap = 0
  cnt_ap = 0
  for cls in range(starting_class, ending_class+1):
      if len(confidence[cls]) == 0:
          continue
      cnt_ap += 1
      if True not in pos_neg[cls]:
          ap = 0
      else:
          ap = average_precision_score(pos_neg[cls], confidence[cls])
      if math.isnan(ap):
          print(pos_neg[cls], confidence[cls])
      sum_ap += ap
      # print("AP for class ", cls, " is ", ap)
      # ap_per_class[cls] = ap_per_class[cls] + ap
      # count[cls] += count[cls] + 1

  if cnt_ap == 0:
      final_map += 1
      # ap_per_class[cls] = ap_per_class[cls] + 1
      # count[cls] += count[cls] + 1
  else:
      final_map +=  sum_ap/cnt_ap
# print(["does not matter", ap_per_class[1]/count[1], ap_per_class[2]/count[2]
return final_map/len(output)
```

```
# def filter_confidence(output, threshold = 0.5):
#   filtered = []
#   for idx in range(len(output)):
#     boxes = output[idx]["boxes"]
#     scores = output[idx]["scores"]
#     labels = output[idx]["labels"]
#     new_boxes = []
#     new_scores = []
#     new_labels = []
#     for box, score, label in zip(boxes, scores, labels):
#       if score > threshold:
#         new_boxes.append(box)
#         new_scores.append(score)
#         new_labels.append(label)
#     if len(new_scores) != 0:
#       new_boxes = torch.stack(new_boxes)
#       new_scores = torch.tensor(new_scores)
#       new_labels = torch.tensor(new_labels)
#     else:
#       new_boxes = torch.tensor([[]])
#       new_scores = torch.tensor([])
#       new_labels = torch.tensor([])
#     temp_filtered = {"boxes": new_boxes, "scores": new_scores, "labels": new_labe
#     filtered.append(temp_filtered)
#   return filtered

# soft nms
def soft_nms_pytorch(dets, box_scores, sigma=0.5, thresh = 0.5, cuda=0):
    """
    Build a pytorch implement of Soft NMS algorithm.
    # Augments
        dets:        boxes coordinate tensor (format:[x1, y1, x2, y2])
        box_scores:  box score tensors
        sigma:       variance of Gaussian function
        thresh:      score thresh
        cuda:        CUDA flag
    # Return
        the index of the selected boxes
    """

    # Indexes concatenate boxes with the last column
    N = dets.shape[0]
    if cuda:
        indexes = torch.arange(0, N, dtype=torch.float).cuda().view(N, 1)
    else:
        indexes = torch.arange(0, N, dtype=torch.float).view(N, 1)
    dets = torch.cat((dets, indexes), dim=1)

    # The order of boxes coordinate is [y1,x1,y2,x2]
    x1 = dets[:, 0]
    y1 = dets[:, 1]
    x2 = dets[:, 2]
    y2 = dets[:, 3]
    scores = box_scores
    areas = (x2 - x1 + 1) * (y2 - y1 + 1)

    for i in range(N):
        # intermediate parameters for later parameters exchange
        tscore = scores[i].clone()
        pos = i + 1

        if i != N - 1:
            maxscore, maxpos = torch.max(scores[pos:], dim=0)
            if tscore < maxscore:
                dets[i], dets[maxpos.item() + i + 1] = dets[maxpos.item() + i + 1]
                scores[i], scores[maxpos.item() + i + 1] = scores[maxpos.item() + 
                areas[i], areas[maxpos + i + 1] = areas[maxpos + i + 1].clone(), a

        # IoU calculate
        yy1 = np.maximum(dets[i, 0].to("cpu").detach().numpy(), dets[pos:, 0].to("
        xx1 = np.maximum(dets[i, 1].to("cpu").detach().numpy(), dets[pos:, 1].to("
        yy2 = np.minimum(dets[i, 2].to("cpu").detach().numpy(), dets[pos:, 2].to("
        xx2 = np.minimum(dets[i, 3].to("cpu").detach().numpy(), dets[pos:, 3].to("

        w = np.maximum(0.0, xx2 - xx1 + 1)
        h = np.maximum(0.0, yy2 - yy1 + 1)
        inter = torch.tensor(w * h).cuda() if cuda else torch.tensor(w * h)
        ovr = torch.div(inter, (areas[i] + areas[pos:] - inter))

        # Gaussian decay
        weight = torch.exp(-(ovr * ovr) / sigma)
        scores[pos:] = weight * scores[pos:]

    # select the boxes and keep the corresponding indexes
    keep = dets[:, 4][scores > thresh].int()

    return keep



def batch_nms_confidence_filter(ls, hard = True, conf_threhold = 0.5):
  for idx in range(len(ls)):

    if hard: # does not give confidence filtered output
      ls_index = torchvision.ops.batched_nms(boxes = ls[idx]["boxes"], scores = ls
      confidence_filtered_ls_index = []
      for index in ls_index:
        if ls[idx]["scores"][index] > conf_threhold:
          confidence_filtered_ls_index.append(index)
      ls_index = confidence_filtered_ls_index
    else:    # gives confidence filtered output
      ls_index = soft_nms_pytorch(ls[idx]["boxes"], ls[idx]["scores"], thresh = co

    temp_ls = {"boxes":[], "scores":[], "labels":[]}

    for index in ls_index:
      temp_ls["boxes"].append(ls[idx]["boxes"][index])
      temp_ls["scores"].append(ls[idx]["scores"][index])
      temp_ls["labels"].append(ls[idx]["labels"][index])
    if len(temp_ls["scores"]) > 0:
      temp_ls["boxes"] = torch.stack(temp_ls["boxes"])
      temp_ls["scores"] = torch.tensor(temp_ls["scores"])
      temp_ls["labels"] = torch.tensor(temp_ls["labels"])
    else:
      temp_ls["boxes"] = torch.tensor([[]])
      temp_ls["scores"] = torch.tensor([])
      temp_ls["labels"] = torch.tensor([])
    ls[idx] = temp_ls
  return ls

def confusion_matrix(ls, targets_model, TP, FP, FN, TN, iou_threshold = iou_thresh
      for x in range(len(ls)):
        predicted_scores = ls[x]["scores"].detach().to("cpu").numpy()
        predicted_labels = ls[x]["labels"].detach().to("cpu").numpy()
        predicted_boxes = ls[x]["boxes"].detach().to("cpu").numpy()
        # ground_scores = targets_model[x]["scores"].detach().to("cpu").numpy()
        ground_labels = targets_model[x]["labels"].detach().to("cpu").numpy()
        ground_boxes =  targets_model[x]["boxes"].detach().to("cpu").numpy()
        for index, (predicted_box, predicted_label) in enumerate(zip(predicted_box
          for idx, (ground_box, ground_label) in enumerate(zip(ground_boxes, groun
            if predicted_label == ground_label and iou(predicted_box, ground_box) >
              TP += 1
              ground_labels[idx] = -1
              predicted_labels[index] = -2
              break

        not_counted = 0
        for label in ground_labels:
          if label != -1:
            not_counted += 1
        FN += not_counted

        false_counted = 0
        for label in predicted_labels:
          if label != -2:
            false_counted += 1
        FP += false_counted
      return (TP, FP, FN, TN)
```

model loading

```
  # my_model = torch.load("/content/drive/MyDrive/SSD detection/SSD detection_3class
  # model = my_model
  # model.eval()


  my_model = torch.load("/content/drive/MyDrive/SSD detection/models/SSD detection_3
  model = my_model
  model.eval()
```

```
    SSD(
  (backbone): SSDLiteFeatureExtractorMobileNet(
    (features): Sequential(
      (0): Sequential(
        (0): ConvNormActivation(
          (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1,
          (1): BatchNorm2d(16, eps=0.001, momentum=0.03, affine=True, track_
          (2): Hardswish()
        )
        (1): InvertedResidual(
          (block): Sequential(
            (0): ConvNormActivation(
              (0): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding
              (1): BatchNorm2d(16, eps=0.001, momentum=0.03, affine=True, t
              (2): ReLU(inplace=True)
            )
            (1): ConvNormActivation(
              (0): Conv2d(16, 16, kernel_size=(1, 1), stride=(1, 1), bias=Fa
              (1): BatchNorm2d(16, eps=0.001, momentum=0.03, affine=True, t
            )
          )
        )
        (2): InvertedResidual(
          (block): Sequential(
            (0): ConvNormActivation(
              (0): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1), bias=Fa
              (1): BatchNorm2d(64, eps=0.001, momentum=0.03, affine=True, t
              (2): ReLU(inplace=True)
            )
            (1): ConvNormActivation(
              (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding
              (1): BatchNorm2d(64, eps=0.001, momentum=0.03, affine=True, t
              (2): ReLU(inplace=True)
            )
            (2): ConvNormActivation(
              (0): Conv2d(64, 24, kernel_size=(1, 1), stride=(1, 1), bias=Fa
              (1): BatchNorm2d(24, eps=0.001, momentum=0.03, affine=True, t
            )
          )
        )
        (3): InvertedResidual(
          (block): Sequential(
            (0): ConvNormActivation(
              (0): Conv2d(24, 72, kernel_size=(1, 1), stride=(1, 1), bias=Fa
              (1): BatchNorm2d(72, eps=0.001, momentum=0.03, affine=True, t
              (2): ReLU(inplace=True)
            )
            (1): ConvNormActivation(
              (0): Conv2d(72, 72, kernel_size=(3, 3), stride=(1, 1), padding
              (1): BatchNorm2d(72, eps=0.001, momentum=0.03, affine=True, t
              (2): ReLU(inplace=True)
            )
            (2): ConvNormActivation(
              (0): Conv2d(72, 24, kernel_size=(1, 1), stride=(1, 1), bias=Fa
              (1): BatchNorm2d(24, eps=0.001, momentum=0.03, affine=True, t
            )
          )
```

```
# vaibhav_model = torch.load("/content/drive/MyDrive/SSD detection/vaibhav_model_d
# model = vaibhav_model
# model.eval()
```

mAP

```
model.eval()
map_class_all = 0
map_class1 = 0
map_class2 = 0
map_class3 = 0
updates = 0
for dic in tqdm(val_loader):
    images_model = []
    targets_model = []
    for x in range(len(dic)):
        images_model.append(dic[x]['image'].float())
        dictionary = {'boxes': dic[x]['bbox'], 'labels': dic[x]['label']}
        targets_model.append(dictionary)
    ls = model.forward(images_model)
    a = ls
    b = targets_model
    ls = batch_nms_confidence_filter(ls, hard = True)
    map_class1 += mean_average_precision(ls, targets_model, iou_threshold, 1,1)
    map_class2 += mean_average_precision(ls, targets_model, iou_threshold, 2,2)
    map_class3 += mean_average_precision(ls, targets_model, iou_threshold, 3,3)
    map_class_all += mean_average_precision(ls, targets_model, iou_threshold, 1,3)
    # confusion_matrix
    # map = map + c
    updates += 1
print("For class 1 map:{} | For class 2 map : {} | For class 3 map : {}".format(ma
print("Overall map :{}".format(map_class_all/updates))
```

CONFUSION MATRIX

```
model.eval()
map = 0
updates = 0
val_loader_idx = 0

TP = 0
FP = 0
FN = 0
TN = 0
```

```
for dic in tqdm(val_loader):
    images_model = []
    targets_model = []
    for x in range(len(dic)):
        images_model.append(dic[x]['image'].float())
        dictionary = {'boxes': dic[x]['bbox'], 'labels': dic[x]['label']}
        targets_model.append(dictionary)
    ls = model.forward(images_model)
    ls = batch_nms_confidence_filter(ls, hard = True)
    TP, FP, FN, TN = confusion_matrix(ls, targets_model, TP, FP, FN, TN, iou_thresh

print("TP = {} | FP = {}".format(TP, FP))
print("----------------")
print("FN = {} | TN = {}".format(FN, "Not defined"))
```

```
# model.eval()
# map = 0
# updates = 0
# val_loader_idx = 0
# for dic in tqdm(val_loader):
#     images_model = []
#     targets_model = []
#     for x in range(len(dic)):
#         images_model.append(dic[x]['image'].float())
#         dictionary = {'boxes': dic[x]['bbox'], 'labels': dic[x]['label']}
#         targets_model.append(dictionary)
#     ls = model.forward(images_model)
#     ls = batch_nms_confidence_filter(ls, hard = True)
#     for x in range(len(ls)):
#       z = dic[x]['image']
#       z[0] = (z[0]*0.2736+0.4662)*255
#       z[1] = (z[1]*0.2650+0.4279)*255
#       z[2] = (z[2]*0.2774+0.3946)*255
#       z = z.cpu().detach().numpy()
#       temp_img = copy.deepcopy(z)
#       temp_img = np.array(temp_img, dtype='uint8')
#       temp_img1 = np.zeros((320, 320, 3), dtype="uint8")
#       for i in range(320):
#         for j in range(320):
#           temp_img1[j][i] = (temp_img[0][j][i], temp_img[1][j][i], temp_img[2][j
#       # print(temp_img)
#       temp_img1 = cv2.cvtColor(temp_img1, cv2.COLOR_BGR2RGB)
#       scores = ls[x]["scores"].detach().to("cpu").numpy()
#       labels = ls[x]["labels"].detach().to("cpu").numpy()
#       boxes = ls[x]["boxes"].detach().to("cpu").numpy()
#       boxes = np.array(boxes, dtype='int')
#       for label, box in zip(labels, boxes):
#         color = None
#         if label == 1:
#           color = (255, 0, 0)
#         elif label == 2:
#           color = (0, 255, 0)
#         elif label == 3:
#           color = (0, 0, 255)
#         # print(temp_img)
#         # print(box)
#         temp_img1 = cv2.rectangle(temp_img1, (box[0], box[1]), (box[2], box[3]),
#       cv2_imshow(temp_img1)
```