BME646 and ECE60146: Homework 5

# Mykhailo Tsysin

# Mar 6, 2023

In this homework, we will look more closely into how to design a convolutional neural network for object detection and localization and how various aspects change the performance. For a full .py solution, please look at the corresponding file.

# Task 3.1

We incorporate some dataset downloading logic into the constructor of the dataset. If the “download” flag is turned on, the constructor will erase everything from the data folder and re-download random pictures from the COCO dataset. The “verify” flag serves as a sanity check if the correct number of images is downloaded. The pre-selected images are then filtered based on the required criteria for box size and for classes present. After this, we create a separate lightweight annotation JSON file where we store the leading class and leading bounding box. The resulting sample outputs are:

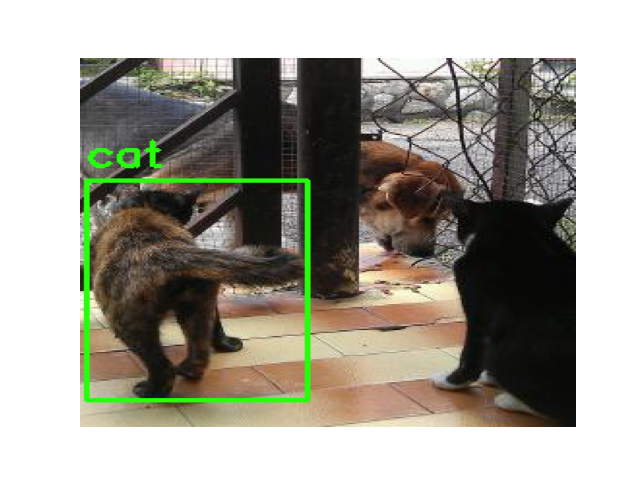
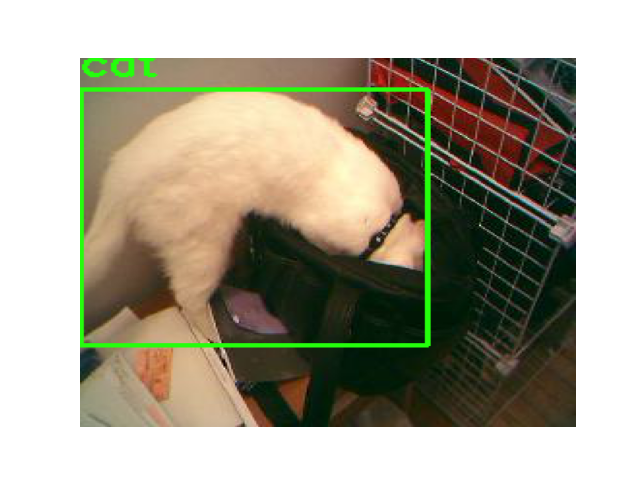
Training:

A picture containing text

Description automatically generatedA cat lying on its back

Description automatically generated with medium confidence

Validation:



We can clearly see that the dataset correctly represents the source data from the COCO dataset.

The code for the dataset is:

class COCODataset(torch.utils.data.Dataset):

"""Iteration of dataset for HW4"""

def \_\_init\_\_ (self,

root,

categories\_list,

num\_train\_min = 1500,

num\_train\_max = 2000,

num\_val\_min = 500,

num\_val\_max = 500,

train = True,

exclusive = True,

clear = False,

transform = None,

scale\_bbox = False,

augmentation = None,

download = False,

verify = False,

):

super ().\_\_init\_\_()

# Obtain meta information (e.g. list of file names)

# Initialize data augmentation transforms, etc.

self.transform = transform

self.scale\_bbox = scale\_bbox

self.augmentation = augmentation

self.root = root

self.num\_min = num\_train\_min if train else num\_val\_min

self.num\_max = num\_train\_max if train else num\_val\_max

self.categories\_list = categories\_list

if download:

# Download dataset

# Get COCO objecct

self.dataType='train2014' if train else 'val2014'

annFile='{}/annotations/instances\_{}.json'.format(self.root,self.dataType)

if os.path.exists(annFile):

coco=COCO(annFile)

else:

raise ValueError(f"Please download the annotation files into {annFile}")

self.catIds = coco.getCatIds(catNms=categories\_list)

self.catIds\_to\_category = {cat\_id: i for i, cat\_id in enumerate(self.catIds)}

# If clear, clear the dataset

if clear:

path = os.path.join(self.root, "data", "train" if train else "val")

if os.path.exists(path):

shutil.rmtree(path)

if not os.path.exists(path):

os.makedirs(path)

# Download images in the dataset

print("CAT IDs ", self.catIds)

imgIds = list(set(sum([coco.getImgIds(catIds=cat\_id) for cat\_id in self.catIds], [])))

# imgIds=coco.getImgIds(catIds=self.catIds)

print("img\_ids ", len(imgIds))

random.shuffle(imgIds) # Load in random order

images = coco.loadImgs(imgIds)

count = 0 # count of downloaded images

im\_iter = iter(images)

print(len(images))

# Create custom lightweight annotation file.

self.annotation = {}

# Download status bar

status\_bar = tqdm.tqdm(total=self.num\_max, desc='LOADED IMAGES', position=0)

status\_bar\_total\_img = tqdm.tqdm(total=len(images), desc='PROCESSED IMAGES', position=1)

while count < self.num\_max:

im = next(im\_iter, -1)

# Check if there is something in the iterator

if im == -1 and count < self.num\_min:

raise StopIteration("We've ran out of images, target not reached")

elif im == -1:

break

# Get annotations for objects

annIds = coco.getAnnIds(imgIds = [im['id']])

anns = coco.loadAnns(annIds)

max\_bbox\_area = 0

max\_box\_area\_cat\_id = None

max\_bbox = None

# Check if there is a dominant object:

for ann in anns:

# bbox\_area = ann["bbox"][2] \* ann["bbox"][3] # W \* H

bbox\_area = ann["area"] # W \* H

# print(ann["bbox"], ann["area"])

# assert ann["bbox"][2] \* ann["bbox"][3] >= ann["area"]

if bbox\_area > max\_bbox\_area:

max\_bbox\_area = bbox\_area

max\_box\_area\_cat\_id = ann["category\_id"]

max\_bbox = ann["bbox"]

# if ann["category\_id"] in self.catIds:

# img\_annotation[ann["category\_id"]] = ann["bbox"]

img\_annotation = [max\_box\_area\_cat\_id, max\_bbox]

# Accept box if the bounding box is maximal, big enough, and comes from correct category

if max\_bbox[2] \* max\_bbox[3] > MIN\_W \* MIN\_H and max\_box\_area\_cat\_id in self.catIds: ############# CHANGES

im\_pil, orig\_shape = self.get\_img\_from\_url(im['coco\_url'])

save\_path = os.path.join(self.root, "data", "train" if train else "val", str(im['id'])+".jpg")

im\_pil.save(save\_path)

w, h = orig\_shape

img\_annotation[1][0] = img\_annotation[1][0] \* 256 // w

img\_annotation[1][1] = img\_annotation[1][1] \* 256 // h

img\_annotation[1][2] = img\_annotation[1][2] \* 256 // w

img\_annotation[1][3] = img\_annotation[1][3] \* 256 // h

img\_annotation[0] = self.catIds\_to\_category[img\_annotation[0]]

self.annotation[im['id']] = img\_annotation

count += 1

status\_bar.update(1)

status\_bar\_total\_img.update(1)

print(f"Loaded {count} images")

with open(os.path.join(self.root, "data", "data\_" + ("train" if train else "val") + ".json"), "w") as outfile:

json.dump(self.annotation, outfile)

if verify:

"""Verify if the dataset has required number of pictures and that the number is"""

path = os.path.join(self.root, "data", "train" if train else "val")

images\_in\_folder = [cat for cat in os.listdir(path) if not cat.startswith('.')]

if len(images\_in\_folder) < self.num\_min or len(images\_in\_folder) > self.num\_max:

raise ValueError(f"Wrong number of pictures: {len(images\_in\_folder)}")

for img in images\_in\_folder:

if not os.path.isfile(os.path.join(path, img)):

raise ValueError("Sub-folders present")

print("Verification Successful")

# Now, assuming we have everything downloaded and allocated in folders

self.path = os.path.join(self.root, "data", "train" if train else "val")

annot\_file\_name = "data\_" + ("train" if train else "val") + ".json"

with open(os.path.join(self.root, "data", annot\_file\_name), "r") as annot\_file:

self.annotation = json.load(annot\_file)

self.img\_list = os.listdir(os.path.join(self.path))

self.img\_list = [file.split('.')[0] for file in self.img\_list if not file.startswith('.') and file.endswith('.jpg')]

def \_\_len\_\_ (self):

# Return the total number of images

return len(self.img\_list)

def \_\_getitem\_\_ (self, index):

# Read an image at index

# Return the tuple : ( augmented tensor , integer label )

# Get category:

img\_index = self.img\_list[index]

category, bbox = self.annotation[img\_index]

bbox = torch.tensor(bbox).float()

im = Image.open(os.path.join(self.path, img\_index + '.jpg'))

if self.transform:

im = self.transform(im)

im = im.to(dtype=torch.float64)

if self.augmentation:

im = self.augmentation(im)

if self.scale\_bbox:

bbox /= 256.0

return im, category, bbox

@staticmethod

def get\_img\_from\_url(url):

pass

# Download the image from the URL

with urllib.request.urlopen(url) as url\_response:

img\_data = url\_response.read()

# Resize image:

im = Image.open(BytesIO(img\_data))

if im.mode != "RGB":

im = im.convert(mode = "RGB")

size = im.size

im = im.resize((256,256), Image.BOX)

return im, size

# Task 3.2

The neural network is presented here. Essentially we’re using the existing skeleton code form the hw description and populating it with implemented ResNet block. The two heads consist of 2 linear layers with reasonable number of connections.

class ResnetBlock(nn.Module):

"""

Inspired by the original implementation in pytorch github

"""

expansion: int = 1

def \_\_init\_\_(

self,

inplanes: int,

outplanes: int,

) -> None:

super().\_\_init\_\_()

norm\_layer = nn.BatchNorm2d

self.conv1 = nn.Conv2d(inplanes, outplanes, kernel\_size=3, padding=1)

self.bn1 = norm\_layer(outplanes)

self.relu = nn.ReLU(inplace=True)

self.conv2 = nn.Conv2d(outplanes, outplanes, kernel\_size=3, padding=1)

self.bn2 = norm\_layer(outplanes)

def forward(self, x):

identity = x

out = self.conv1(x)

out = self.bn1(out)

out = self.relu(out)

out = self.conv2(out)

out = self.bn2(out)

out += identity

out = self.relu(out)

return out

IM\_SIZE = 256

class HW5Net(nn.Module):

"""

Resnet-based encoder that consists of a few downsampling + several Resnet blocks as the backbone and two prediction heads.

"""

def \_\_init\_\_(self, input\_nc, output\_nc, ngf=8, n\_blocks=4, classes=3):

""" Parameters:

input\_nc (int) -- the number of channels in input images

output\_nc (int) -- the number of channels in output images)

ngf (int) -- the number of filters first conv layer

n\_blocks (int) -- teh number of ResNet blocks

"""

assert (n\_blocks >= 0)

super(HW5Net, self).\_\_init\_\_()

# The first conv layer

model = [

nn.ReflectionPad2d(3),

nn.Conv2d(input\_nc, ngf, kernel\_size=7, padding=0),

nn.BatchNorm2d(ngf),

nn.ReLU(True)

]

# Add downsampling layers

n\_downsampling = 4

for i in range(n\_downsampling):

mult = 2 \*\* i

model += [

nn.Conv2d(ngf \* mult, ngf \* mult \* 2, kernel\_size=3, stride=2, padding=1),

nn.BatchNorm2d(ngf \* mult \* 2),

nn.ReLU(True)

]

# Add your own ResNet blocks

mult = 2 \*\* n\_downsampling

for i in range(n\_blocks):

model += [ResnetBlock(ngf \* mult, ngf \* mult)]

# Generate final model

self.model = nn.Sequential(\*model)

# The classification head

class\_head = [

nn.MaxPool2d(2, 2),

nn.Flatten(),

nn.Linear(ngf \* IM\_SIZE \* IM\_SIZE // mult // 4, 64),

nn.Dropout(0.5),

nn.LeakyReLU(0.1),

nn.Linear(64, classes)

]

self.class\_head = nn.Sequential(\*class\_head) # The bounding box regression head

# Localization head

bbox\_head = [

nn.MaxPool2d(2, 2),

nn.Flatten(),

nn.Linear(ngf \* IM\_SIZE \* IM\_SIZE // mult // 4, 64),

nn.Dropout(0.5),

nn.LeakyReLU(0.1),

nn.Linear(64, 4)

]

self.bbox\_head = nn.Sequential(\*bbox\_head)

def forward(self, input):

ft = self.model(input)

cls = self.class\_head(ft)

bbox = self.bbox\_head(ft)

return cls, bbox

For the training loop we also create 2 classes of loss: MSE loss and complete IOU loss which uses a built-in function with some pre-processing.

def train(net, save = False):

# Choose device

if torch.cuda.is\_available()== True:

device = torch.device("cuda:0")

else:

device = torch.device("cpu")

net.train()

# Create transform

transform = tvt.Compose([tvt.ToTensor(), tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch = 10

train\_dataset = COCODataset(

root=ROOT,

categories\_list=['bus', 'cat', 'pizza'],

num\_train\_min=2000,

num\_train\_max=4000,

num\_val\_min=1000,

num\_val\_max=3000,

download = False,

verify = True,

train = True,

transform=transform,

scale\_bbox = True,

)

train\_data\_loader = torch.utils.data.DataLoader(dataset = train\_dataset,

batch\_size = batch,

shuffle = True,

num\_workers = 0)

net = net.to(device)

criterion\_class = torch.nn.CrossEntropyLoss()

criterion\_localization = LocLoss()

# criterion\_localization = torch.nn.MSELoss(reduction="sum")

optimizer = torch.optim.Adam(

net.parameters(),

lr=1e-3,

betas=(0.9, 0.99)

)

losses = []

losses\_class = []

losses\_loc = []

epochs = 5

file\_log = tqdm.tqdm(total=0, position=1, bar\_format='{desc}')

outer = tqdm.tqdm(total=epochs, desc='Epochs', position=0)

for epoch in range(epochs):

running\_loss = 0.0

running\_loss\_class = 0.0

running\_loss\_loc = 0.0

inner = tqdm.tqdm(total=len(train\_data\_loader), desc='Batches', position=0)

for i, data in enumerate(train\_data\_loader):

inputs, labels\_classes, labels\_bboxes = data

inputs = inputs.to(device)

labels\_classes = labels\_classes.to(device)

labels\_bboxes = labels\_bboxes.to(device)

optimizer.zero\_grad()

output\_classes, output\_bboxes = net(inputs)

loss\_class = criterion\_class(output\_classes, labels\_classes)

loss\_loc = criterion\_localization(output\_bboxes, labels\_bboxes)

loss = loss\_class + loss\_loc

loss.backward()

optimizer.step()

running\_loss += loss.item()

running\_loss\_class += loss\_class.item()

running\_loss\_loc += loss\_loc.item()

if (i+1) % 100 == 0:

file\_log.set\_description\_str(

"[epoch: %d, batch: %5d] loss: %.3f" % (epoch + 1, i + 1, running\_loss / 100)

)

losses.append(running\_loss / 100)

losses\_class.append(running\_loss\_class/100)

losses\_loc.append(running\_loss\_loc/100)

running\_loss = 0.0

running\_loss\_class = 0.0

running\_loss\_loc = 0.0

# print("Labels bboxes", labels\_bboxes)

# print("Labels classes", labels\_classes)

# print("OUT bboxes", output\_bboxes)

# print("OUT classes", output\_classes)

inner.update(1)

outer.update(1)

if save:

torch.save(net.state\_dict(), ROOT+'/model')

return losses, losses\_class, losses\_loc

def iou(true\_box, pred\_box):

true = torch.cat((true\_box[..., :2], true\_box[..., :2] + true\_box[..., 2:4]), -1).unsqueeze(0)

pred = torch.cat((pred\_box[..., :2], pred\_box[..., :2] + pred\_box[..., 2:4]), -1).unsqueeze(0)

return torchvision.ops.box\_iou(true, pred)[0, 0]

def val(net, load\_path=None):

# Choose device

if torch.cuda.is\_available() == True:

device = torch.device("cuda:0")

else:

device = torch.device("cpu")

if load\_path:

net.load\_state\_dict(torch.load(load\_path))

net.eval()

# Create transform

transform = tvt.Compose([tvt.ToTensor(), tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch = 10

val\_dataset = COCODataset(

root=ROOT,

categories\_list=['bus', 'cat', 'pizza'],

num\_train\_min=2000,

num\_train\_max=4000,

num\_val\_min=1000,

num\_val\_max=3000,

download = False,

verify = True,

train = False,

transform=transform

)

val\_data\_loader = torch.utils.data.DataLoader(dataset = val\_dataset,

batch\_size = batch,

shuffle = True,

num\_workers = 0)

test\_loss, correct = 0, 0

criterion\_class = torch.nn.CrossEntropyLoss()

criterion\_localization = LocLoss()

size = len(val\_data\_loader.dataset)

true\_labels = []

pred\_labels = []

test\_iou = 0

with torch.no\_grad():

for i, data in tqdm.tqdm(enumerate(val\_data\_loader)):

inputs, labels\_classes, labels\_bboxes = data

inputs = inputs.to(device)

labels\_classes = labels\_classes.to(device)

labels\_bboxes = labels\_bboxes.to(device)

# inputs, labels = data

# inputs = inputs.to(device)

# labels = labels.to(device)

# outputs = net(inputs)

output\_classes, output\_bboxes = net(inputs)

for i in range(output\_bboxes.size()[0]):

test\_iou += iou(output\_bboxes[i, ...], labels\_bboxes[i, ...])

test\_loss += criterion\_class(output\_classes, labels\_classes).item() + \

criterion\_localization(output\_bboxes, labels\_bboxes).item()

correct += (output\_classes.argmax(1) == labels\_classes).type(torch.float).sum().item()

pred\_labels.extend(output\_classes.argmax(1).view(-1).numpy())

true\_labels.extend(labels\_classes.view(-1).numpy())

test\_iou /= size

test\_loss /= size #batch

correct /= size

print(f"Test Error: \n Accuracy: {(100\*correct):>0.1f}%, Avg loss: {test\_loss:>8f} \n")

print(f"Test IOU Error: {test\_iou}\n")

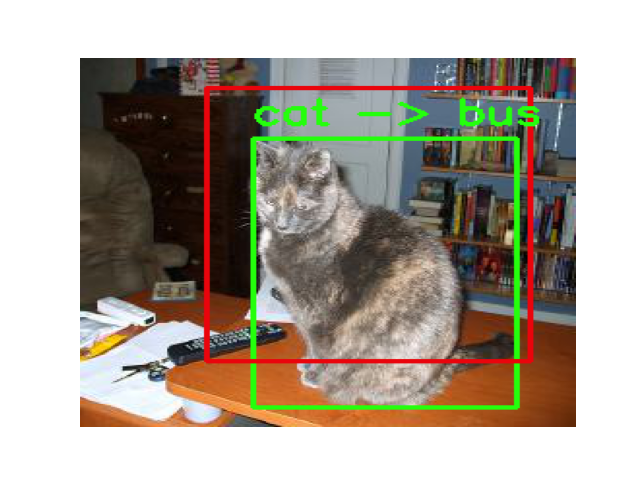
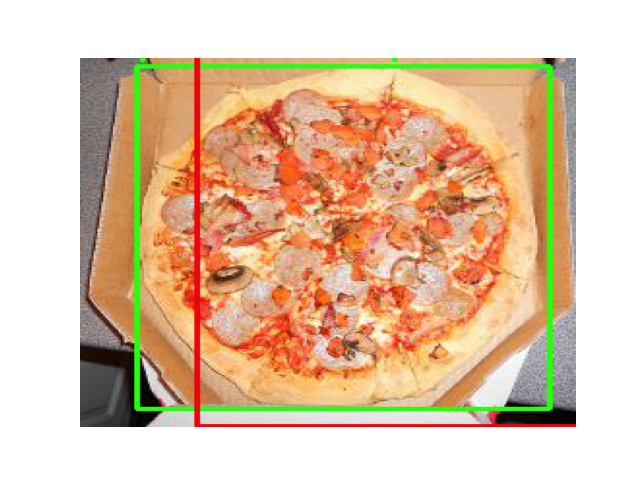
return confusion\_matrix(true\_labels, pred\_labels)

The final accuracy of the model after 10 epochs is close to 91.2% with the following conusion matrix:

Chart

Description automatically generated

The sample IOU outputs are given here. The average accuracy is 0.52 IOU:



(wrongly classified…)

MSE loss vs Complete IOU loss didn’t make much change in resulting accuracy: 0.49 and 0.52 average IOU respectively.

The pizza detector developed in this homework has a pretty good accuracy of classification. Still, the localization has a lot of errors. It might be caused by the fact that multiple objects are present in the image and sometimes the network doesn’t know which one to take. Also, the network architecture is not very robust for different scales.