BME646 and ECE60146: Homework 6

# Mykhailo Tsysin

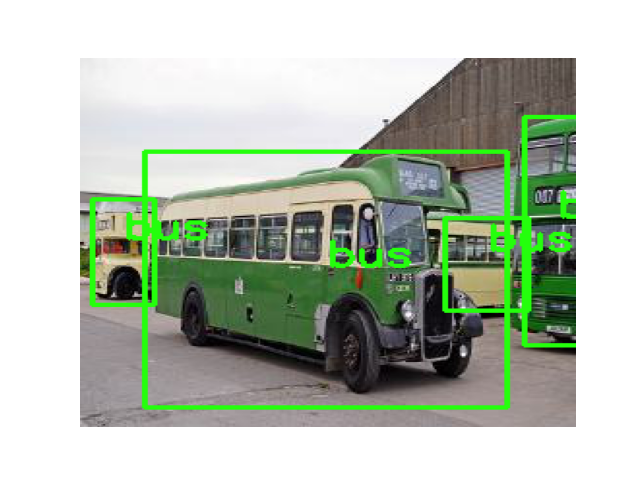
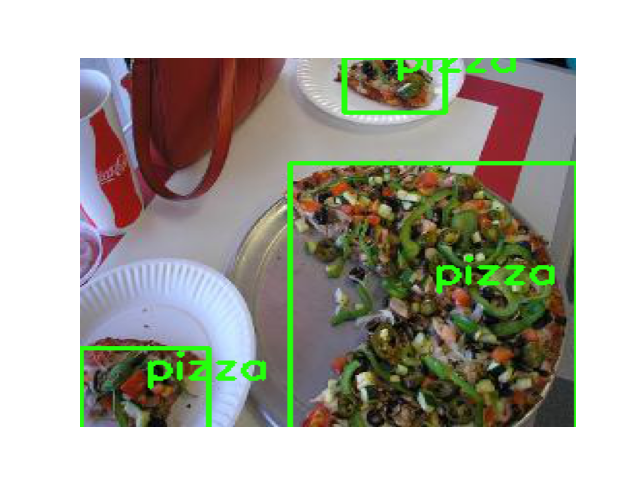
# Mar 20, 2023

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

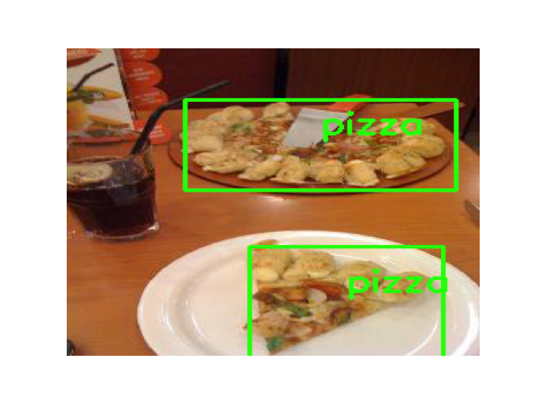
# Task 3.1

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 sizes and for classes present. After this, we create a separate lightweight annotation JSON file where we store the information about every bounding box with corresponding classes. The resulting sample outputs with multiple objects are (note that there are relatively few such images):

Training:

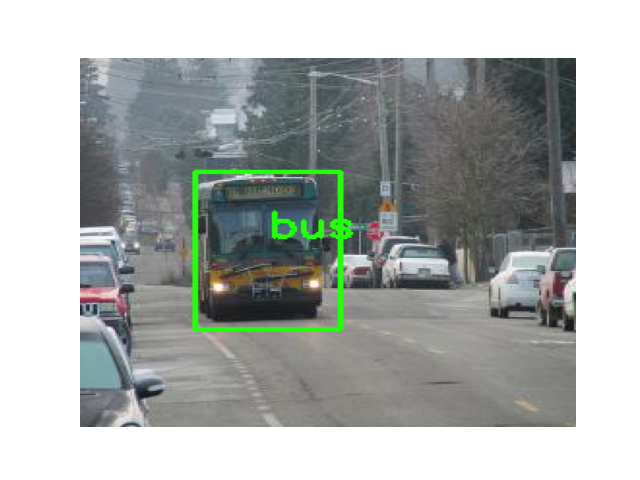






Validation:

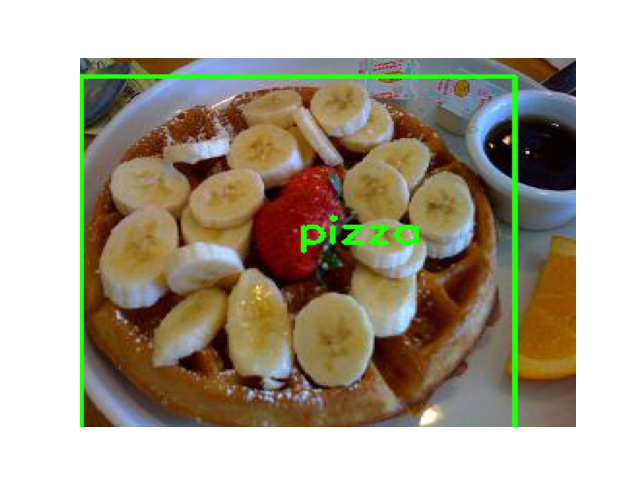
A bus parked in a garage

Description automatically generated with low confidence

A picture containing text, different

Description automatically generatedA picture containing text, cat, indoor, black

Description automatically generated



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

The code for the dataset is provided below. Note that the \_\_getitem\_\_ method outputs directly the required data structure, so we don’t need to do that in the training loop.

MIN\_W = 64

MIN\_H = 64

TARGET\_SIZE = 265

ROOT = "."

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

"""Iteration of dataset for HW4"""

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

root,

categories\_list,

num\_train\_min = 4500,

num\_train\_max = 7000,

num\_val\_min = 2500,

num\_val\_max = 4000,

train = True,

clear = False,

transform = None,

augmentation = None,

download = False,

verify = False,

grid\_size = 32,

return\_raw = False,

anchor\_boxes = 5

):

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

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

# Initialize data augmentation transforms, etc.

self.transform = transform

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

self.grid\_size = grid\_size

self.return\_raw = return\_raw

self.anchor\_boxes = anchor\_boxes

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("Number of unfiltered images: ", 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) # annotation for particular image

max\_bbox\_area = 0

max\_box\_area\_cat\_id = None

max\_bbox = None

img\_annotation = []

# Check if there is a dominant object:

for ann in anns:

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

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

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

obj\_cat\_id = ann["category\_id"]

if obj\_area > MIN\_W \* MIN\_H and obj\_cat\_id in self.catIds:

img\_annotation.append([obj\_cat\_id, ann["bbox"]])

# Accept image if there is something in annotations:

if img\_annotation: ############# 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

# Scale bounding boxes accordingly:

img\_annotation = [

[

self.catIds\_to\_category[ann[0]],

[

ann[1][0] \* TARGET\_SIZE // w,

ann[1][1] \* TARGET\_SIZE // h,

ann[1][2] \* TARGET\_SIZE // w,

ann[1][3] \* TARGET\_SIZE // h,

]

]

for ann in img\_annotation

]

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))

# Remove clutter

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]

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

if self.transform:

im = self.transform(im)

im = im.to(dtype=torch.float32)

if self.augmentation:

im = self.augmentation(im)

annotations = self.annotation[img\_index]

# If return raw, return it

if self.return\_raw:

return im, annotations

# bounding box [top left x position , top left y position , width, height]

#Iterate through annotations to generate a yolo tensor:

S = TARGET\_SIZE // self.grid\_size

A = self.anchor\_boxes

C = len(self.categories\_list)

label = torch.zeros((A, S, S, 5 + C))

for ann in annotations:

# Determine the correct box index

# Get coordiantes:

x\_tl, y\_tl, w, h = ann[1]

x\_center, y\_center = x\_tl + w / 2.0, y\_tl + h / 2.0

cell\_x\_idx = int(min(S-1, x\_center // self.grid\_size))

cell\_y\_idx = int(min(S-1, y\_center // self.grid\_size))

# select anchor box

w\_scale, h\_scale = w / self.grid\_size, h / self.grid\_size

AR = h / w

if AR <= 0.2:

abox\_idx = 0

w\_scale, h\_scale = w / 3, h

if 0.2 < AR <= 0.5:

abox\_idx = 1

w\_scale, h\_scale = w / 2, h

if 0.5 < AR <= 1.5:

abox\_idx = 2

w\_scale, h\_scale = w, h

if 1.5 < AR <= 4.0:

abox\_idx = 3

w\_scale, h\_scale = w, h / 2

if 4.0 < AR:

abox\_idx = 4

w\_scale, h\_scale = w, h / 3

bbox\_scaled = [

x\_center/self.grid\_size - (cell\_x\_idx + 0.5),

y\_center/self.grid\_size - (cell\_y\_idx + 0.5),

math.log(w\_scale/self.grid\_size),

math.log(h\_scale/self.grid\_size)

]

yolo\_vector = torch.FloatTensor([1] + bbox\_scaled + [0] \* C)

if label[abox\_idx, cell\_x\_idx, cell\_y\_idx, 0] == 0:

label[abox\_idx, cell\_x\_idx, cell\_y\_idx, ...] = yolo\_vector

# Set objectness

label[abox\_idx, cell\_x\_idx, cell\_y\_idx, 0] = 1

# Set class

label[abox\_idx, cell\_x\_idx, cell\_y\_idx, 5 + ann[0]] = 1

return im, label.float()

@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

For the testing code, refer to the source files.

# Task 3.2

The neural network is presented here. Essentially, we’re using slightly modified version of the network form the last homework. The final output is reshaped into the following:

#BATCH, #ANCHOR, GRIDX, GRIDY, (#CLASSES + 5)

Corresponding to the required data stricture: for each anchor at each cell we have 3 class indicators and 5 parameters: objectness, and x, y, w, h coordinates scaled appropriately.

class ResnetBlock(nn.Module):

"""

Inspired by the original implementation in pytorch github

"""

expansion: int = 1

def \_\_init\_\_(

self,

inplanes: int,

outplanes: int,

downsample: Union[str, bool] = None

) -> None:

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

norm\_layer = nn.BatchNorm2d

self.downsample = downsample

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

self.bn1 = norm\_layer(outplanes)

self.relu = nn.ReLU()

self.conv2 = nn.Conv2d(outplanes, outplanes, kernel\_size=3, padding=1, stride = (1 if downsample in (False, None) else 2))

self.bn2 = norm\_layer(outplanes)

if self.downsample == True:

self.downsample = nn.Conv2d(inplanes, outplanes, kernel\_size=1, stride=2)

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)

if self.downsample:

identity = self.downsample(identity)

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, ngf=8, n\_blocks=4, classes=3, anchors = 5):

""" 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

"""

self.classes = classes

self.anchors = anchors

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\_1 = 3

mult = 1

for \_ in range(n\_downsampling\_1):

model += [

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

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

nn.ReLU(True)

]

mult\*=2

# Add your own ResNet blocks

for i in range(n\_blocks):

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

# Add downsampling layers with ResNet

n\_downsampling\_2 = 2

for \_ in range(n\_downsampling\_2):

model += [

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

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

# nn.ReLU(True),

ResnetBlock(ngf \* mult, ngf \* mult \* 2, downsample=True)

]

mult\*=2

# Generate final model

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

# Head for generating the output of the network

head = [

nn.Conv2d(ngf \* mult, (classes + 5) \* anchors, kernel\_size=1, stride=1, padding=0)

]

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

def forward(self, input):

ft = self.model(input)

# print("pre ", ft[0, 0, 0,...])

out = self.head(ft).view(-1, self.anchors, 8, 8, 5 + self.classes)

return out

The number of downsampling ResNet blocks can be modified to accommodate different numbers of cells in the grid.

We also pack loss logic into a separate module: It returns all 3 losses for easier logging. We can also add coefficients for different parts of the loss. For example, we weigh the boxes with the object more than the boxes without objects. Otherwise, all objectness scores tend to be very low.

EPS = 1e-6

class YOLOLoss(nn.Module):

"""

Loss for YOLO network, implemented based on the original paper.

Input parameters:

split\_grids (int): number of cells in each direction (S)

num\_bboxes (int): number of bounding boxes to predict per cell

num\_classes (int): number of dataset classes

lambda\_coord (int): weight for location loss

lambda\_noobj (int): weight for the case where there is no object in the cell.

"""

def \_\_init\_\_(self, split\_grids=8, num\_anchor=5, num\_classes=3, lambda\_coord=5, lambda\_noobj=0.5) -> None:

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

self.S = split\_grids

self.A = num\_anchor

self.C = num\_classes

self.lambda\_coord = lambda\_coord

self.lambda\_noobj = lambda\_noobj

self.criterion1 = nn.BCELoss() # for the objectness score, applied to all of the boxes

self.criterion2 = nn.MSELoss() # for the position of the bboxes, applied only when there is an object in the box

self.criterion3 = nn.CrossEntropyLoss() # for the classification error, applied only if there is an object

def forward(self, pred, target):

"""

The function takes a prediction tensor of shape (batch\_size, A, S, S, 5 + C)

and target tensor of shape (batch\_size, A, S, S, 5 + C)

The last dimension is organized as [class probabilities, (score, x,y,w,h)]

The output is a single float number -- resulting loss

"""

# print("dtype pred = ", pred.dtype)

# print("dtype pred = ", target.dtype)

# Find indices where there is an object:

Iobj\_i = target[..., 0].bool()

# print("Iobj\_i.size() = ", Iobj\_i.size())

object\_present\_target = target[Iobj\_i]

# print("object\_present\_target.size() = ", object\_present\_target.size())

object\_present\_pred = pred[Iobj\_i]

# print("object\_present\_pred.size() = ", object\_present\_pred.size())

pred\_bce = nn.Sigmoid()(pred[..., 0].unsqueeze(-1))

# print("pred BCEEEEE", torch.max(pred\_bce), torch.min(pred\_bce))

# print("pred BCEEEEE", torch.max(target[..., 0].unsqueeze(-1)), torch.min(target[..., 0].unsqueeze(-1)))

loss1 = self.criterion1(pred\_bce, target[..., 0].unsqueeze(-1)) \

+ 10 \* self.criterion1(nn.Sigmoid()(object\_present\_pred[..., 0].unsqueeze(-1)), object\_present\_target[..., 0].unsqueeze(-1))# Separately for objects

loss2 = self.criterion2(object\_present\_pred[..., 1:5], object\_present\_target[..., 1:5])

loss3 = self.criterion3(object\_present\_pred[..., 5:], object\_present\_target[..., 5:])

# print(loss1, loss2, loss3)

# print(object\_present\_target, object\_present\_pred)

return loss1, loss2, loss3

The training loop is generic with all the functionality offloaded to the corresponding classes:

MIN\_W = 200

MIN\_H = 200

ROOT = "."

LOSS\_COUNT = 50

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'],

download = False,

verify = True,

train = True,

transform=transform,

)

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

batch\_size = batch,

shuffle = True,

num\_workers = 0)

net = net.to(device)

criterion = YOLOLoss()

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

optimizer = torch.optim.Adam(

net.parameters(),

lr=1e-3,

betas=(0.9, 0.99)

)

losses = []

losses\_separate = []

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\_separate = [0.0] \* 3

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

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

inputs, labels = data

inputs = inputs.to(device)

# print(inputs[0, 0, 0,...])

labels = labels.to(device)

optimizer.zero\_grad()

outputs = net(inputs)

# print(outputs[0, 0, 0,...])

batch\_losses = criterion(outputs, labels)

# print(batch\_losses)

loss = sum(batch\_losses)

# print(loss)

loss.backward()

# for param in net.parameters():

# print(param.grad[0,...], param.size())

# break

optimizer.step()

running\_loss += loss.item()

running\_loss\_separate = [curr\_loss + new\_loss.item() for curr\_loss, new\_loss in zip(running\_loss\_separate, batch\_losses)]

if (i+1) % LOSS\_COUNT == 0:

file\_log.set\_description\_str(

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

)

losses.append(running\_loss / LOSS\_COUNT)

losses\_separate.append([el/LOSS\_COUNT for el in running\_loss\_separate])

running\_loss = 0.0

running\_loss\_separate = [0.0] \* 3

# 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\_separate

There is no validation loop this time since we’re just looking at the output.

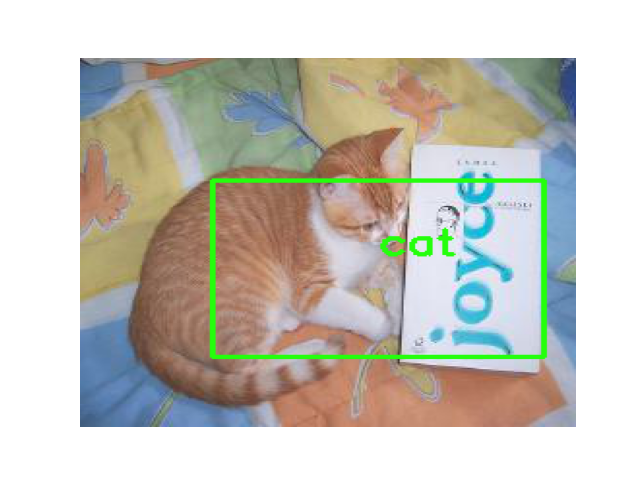
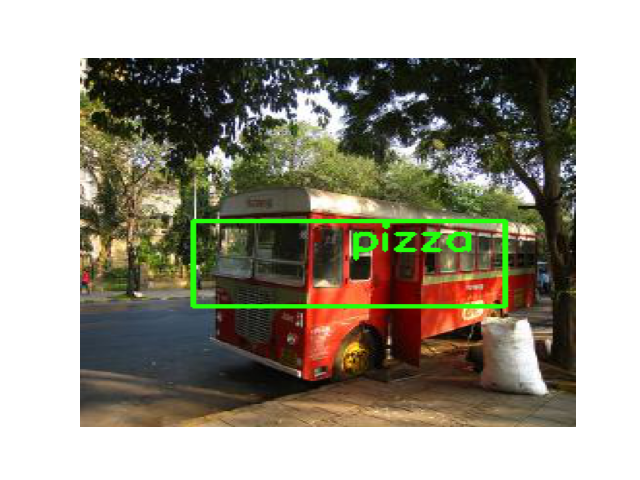
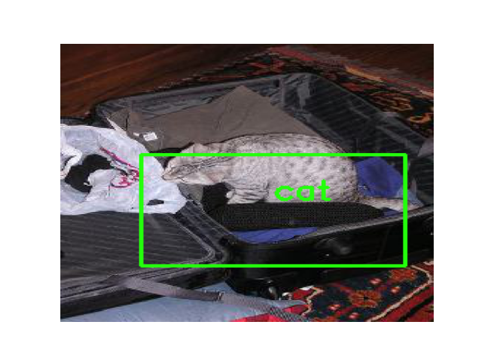
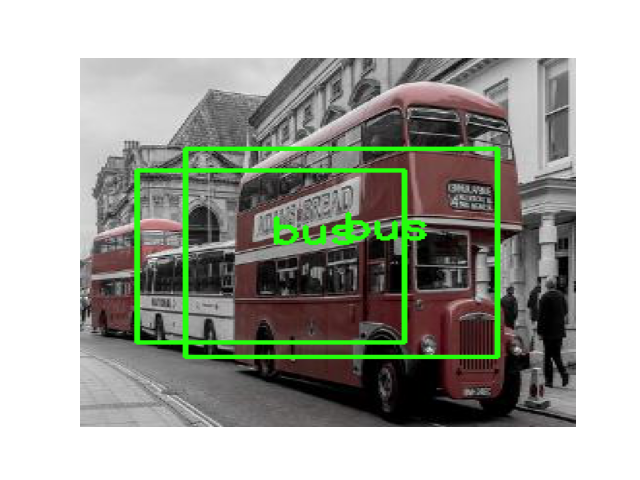
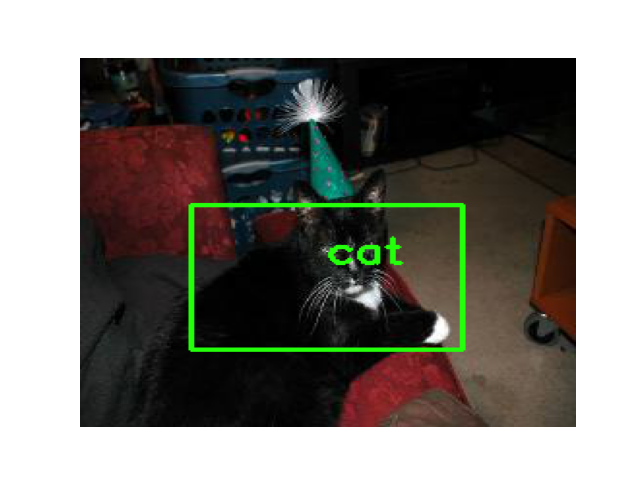
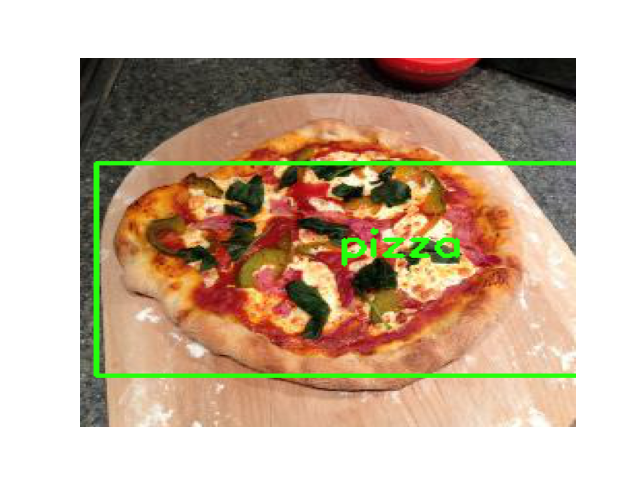
For this homework, we decreased the learning rate and increased the number of epochs. The loss over 100 epochs trained on TITAN X is presented here. The learning rate was scheduled to decrease every 20 epochs in addition to ADAM’s optimization.

Histogram

Description automatically generated

The resulting model is relatively good at identifying object classes but struggles a lot with localizations. The probable reason behind this is the instability of objectness scores, which are not determined very precisely. The introduction of focal loss or further weight tweaking might improve the performance. Also, with more time, it’s possible to further play with the network architecture to create a better feature extractor. As a possibility, we can use pre-trained Darknet53 or a similar network as a feature extractor.

The overall performance is not super great. The model sometimes produces relatively solid output but often misses (I didn’t include true boxes not to obstruct the image, they can be easily inferred from the picture). Also, the number of images that contain more than 1 object is very low, that’s why the performance of the model is relatively poor.

Graphical user interface

Description automatically generated

