Homework 3
AA275: Navigation for Autonomous Systems

Alexandros Tzikas
alextzik@stanford.edu

November 19, 2021

1 Problem 1

In this homework, the emphasis is on Deep Learning using PyTorch. The goal is to build and train neural networks for vehicle classification and segmentation.

1.1

For this part, we will build a simple vehicle classifier, which takes 32×32 RGB images and assigns a label 1 if the image contains a vehicle an 0 otherwise.

1.1.1

The RGB images and the associated ground truth were downloaded from the Virtual KITTI 2 dataset. The classification dataset was generated for "Scene02", which contained 218 positive and 217 negative examples. The required metrics are shown below in the form [R, G, B]:

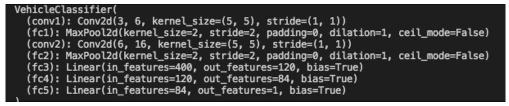
- Mean: $[0.61717, 0.625268, 0.519235]$
- Standard Deviation: $[0.252, 0.244, 0.2677]$

The code for datagen.py is included in Section 3.

1.1.2

The load_data method was completed in classifier_utils .py, by adding the necessary transforms. The dataset was loaded into separate train and test loaders. The code for classifier_utils .py is included in Section 3, but the relevant piece of the code is included below.

```
def load_data(class_dataset_path):
    batch_size = 8
    num_workers = 0
    test_fraction = 0.1

    transform = transforms.Compose([
        transforms.Resize((32,32)),
        transforms.ToTensor(),
        transforms.Normalize(
            mean = [0.61717, 0.6252, 0.5192],
            std = [0.25209, 0.244, 0.2677]
        ),
        transforms.RandomHorizontalFlip(p=0.5)
    ])
```



```
VehicleClassifier(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (fc1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc3): Linear(in_features=400, out_features=120, bias=True)
  (fc4): Linear(in_features=120, out_features=84, bias=True)
  (fc5): Linear(in_features=84, out_features=1, bias=True)
)
```

Figure 1: Model Architecture.

```
    # TODO: Add transforms for preprocessing and data augmentation.
    # You should atleast include:
    # - resize to 32X32 images
    # - Normalize by the mean and std deviation of the dataset
    # - Randomly flip the image horizontally with a probability of 0.5

    dataset = datasets.ImageFolder(class_dataset_path, transform=transform)

    num_train = len(dataset)
    indices = list(range(num_train))
    split = int(np.floor(test_fraction * num_train))
    # TODO: Load the classification dataset into train and test loaders
    np.random.shuffle(indices)

    train_idx = indices[split:]
    train_sampler = SubsetRandomSampler(train_idx)

    test_idx = indices[:split]
    test_sampler = SubsetRandomSampler(test_idx)

    print(len(train_idx))
    print(len(test_idx))

    train_loader = torch.utils.data.DataLoader(
        dataset, batch_size = batch_size, sampler = train_sampler, num_workers = num_workers
    )
    test_loader = torch.utils.data.DataLoader(
        dataset, batch_size = batch_size, sampler = test_sampler, num_workers = num_workers
    )

    return (train_loader, test_loader)
```

1.1.3

The VehicleClassifier neural network architecture class and forward pass were completed. The created architecture can be seen in Figure 1. The code can be found in classifier_utils .py, which is included in Section 3.

1.1.4

The optimizer was completed based on the demo slides, along with the bce_loss and binary_acc methods. The code can be found in classifier_utils .py, which is included in Section 3, but the relevant part is included below.

```
def bce_loss(y_pred, y_target):
    # TODO: compute binary cross entropy loss from NN output y_pred and target y_target
    bceLoss = nn.BCELoss()
    loss = bceLoss(y_pred, y_target)
```

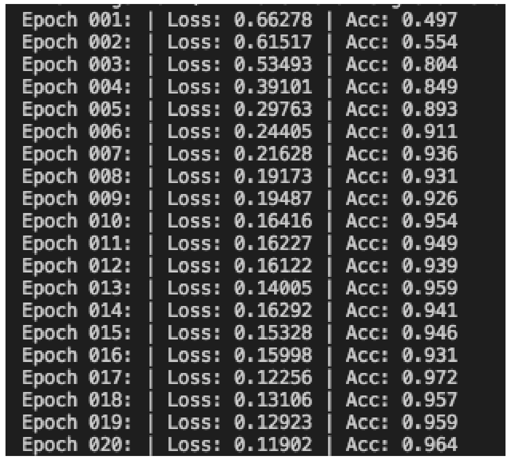
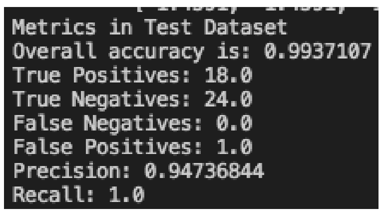
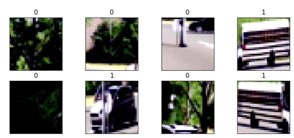

```
Epoch 001: | Loss: 0.66278 | Acc: 0.497
Epoch 002: | Loss: 0.61517 | Acc: 0.554
Epoch 003: | Loss: 0.53493 | Acc: 0.804
Epoch 004: | Loss: 0.39101 | Acc: 0.849
Epoch 005: | Loss: 0.29763 | Acc: 0.893
Epoch 006: | Loss: 0.24405 | Acc: 0.911
Epoch 007: | Loss: 0.21628 | Acc: 0.936
Epoch 008: | Loss: 0.19173 | Acc: 0.931
Epoch 009: | Loss: 0.19487 | Acc: 0.926
Epoch 010: | Loss: 0.16416 | Acc: 0.954
Epoch 011: | Loss: 0.16227 | Acc: 0.949
Epoch 012: | Loss: 0.16122 | Acc: 0.939
Epoch 013: | Loss: 0.14005 | Acc: 0.959
Epoch 014: | Loss: 0.16292 | Acc: 0.941
Epoch 015: | Loss: 0.15328 | Acc: 0.946
Epoch 016: | Loss: 0.15998 | Acc: 0.931
Epoch 017: | Loss: 0.12256 | Acc: 0.972
Epoch 018: | Loss: 0.13106 | Acc: 0.957
Epoch 019: | Loss: 0.12923 | Acc: 0.959
Epoch 020: | Loss: 0.11902 | Acc: 0.964
```

Figure 2: Training results.


```
Metrics in Test Dataset
Overall accuracy is: 0.9937107
True Positives: 18.0
True Negatives: 24.0
False Negatives: 0.0
False Positives: 1.0
Precision: 0.94736844
Recall: 1.0
```

(a) Metrics in Test Dataset.



(b) Batch of Test Outputs.

Figure 3: Performance on Test Set.

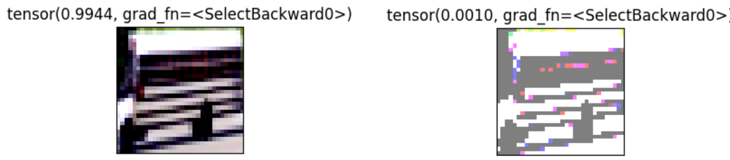
```python
    return loss

def binary_acc(y_pred, y_target):
    # TODO: compute accuracy of the NN output y_pred from target
    y_target
    y_acc = y_pred.clone().detach().requires_grad_(False)
    y_acc[y_pred>0.5] = 1
    y_acc[y_pred<=0.5] = 0
    acc = sum(y_acc==y_target)/len(y_pred)
    return acc

 optimizer = torch.optim.Adam(net.parameters(), lr=0.0001)
```

1.1.5

The train method was implemented and the classifier was trained for 20 epochs. The training results can be seen in Figure 2.

1.1.6

The test method was implemented. The classifier's overall accuracy, confusion matrix, precision and recall on the test dataset are shown in Figure 3a, while a batch of outputs appears in Figure 3b. We can observe that the high accuracy entails high recall and precision, while we notice great performance in the output batch visualized.



(a) Original image. (b) Altered image.

Figure 4: Original and Altered Image Classification.

1.1.7

The trained classifier accurately detects a vehicle in the image shown in Figure 4a. We altered the image using noise = transforms.ColorJitter(brightness=10, contrast=20, saturation=0, hue=0) (altered the brightness and contrast) and we observe in Figure 4b that the classifier misdetects the vehicle present in the image, due to the noise. In general therefore, noise perturbations can impact and degrade the performance of the classifier, which is not desirable for applications in autonomous navigation, which are safety-critical. In order to improve the classifier's performance we can increase its robustness. To do so we can add to the training data images with noise perturbations, in order to make the classifier more robust to slight variations among image inputs. We can also train the classifier by minimizing the loss about the worst perturbation of the example to ensure robustness.

1.2

In this second part, we will train a deep neural network on the Virtual KITTI 2 dataset to detect and segment out vehicles present in images. The Mask R-CNN architecture with a retrained MobileNet v2 as the feature extraction backbone is used.

1.2.1

The VkittiDataset class was completed in detection_utils .py. The code for detection_utils .py is included in Section 3, but the relevant part is included below.

```python
# Define dataset class
class VkittiDataset(Dataset):
    def __init__(self, full_dataset_path, transforms=None):
        self.root = full_dataset_path
        self.transforms = transforms

        # load all image files, sorting them to
        # ensure that they are aligned
        scenes = ['Scene01', 'Scene02', 'Scene06', 'Scene18', 'Scene20']

        self.imgs = []
        for scene in scenes:
            dataset_path = full_dataset_path.format(scene=scene,
            type="rgb")
            for path in os.listdir(dataset_path):
                full_path = os.path.join(dataset_path, path)
                self.imgs.append(full_path)
        self.imgs = list(sorted(self.imgs))

        self.masks = []
        for scene in scenes:
```

```python
            dataset_path = full_dataset_path.format(scene=scene,
            type="instanceSegmentation")
            for path in os.listdir(dataset_path):
                full_path = os.path.join(dataset_path, path)
                self.masks.append(full_path)
        self.masks = list(sorted(self.masks))

    def __getitem__(self, idx):

        img = Image.open(self.imgs[idx]).convert("RGB")     # open image
        and convert to RGB if grayscale
        mask = Image.open(self.masks[idx])    # open mask (no conversion)


        mask = np.array(mask)    # convert mask to np array. Mask has
        size (375, 1242) as the image

        #TODO: find all unique entities present in the image. Remove any
        unwanted classes, such as 0 (background) (ok)
        obj_ids = np.sort(np.unique(mask))

        if obj_ids[0]==0:
            obj_ids = np.delete(obj_ids, 0)

        num_objs = len(obj_ids)

        #TODO: for each object id, create a corresponding binary mask
        # Each mask must be of type bool and true where the object is
        present and false everywhere else (ok)
        masks = np.zeros((num_objs, mask.shape[0], mask.shape[1]))
        for i in range(num_objs):
            masks[i,:,:] = (mask==obj_ids[i])


        # get bounding box coordinates from the masks
        boxes = []
        issmall = np.zeros(num_objs, dtype=bool) # 1 if bounding box is
        smaller than 20X20 pixels

        # TODO: get bounding box coordinates from the masks
        for i in range(num_objs):
            # horizontal_indicies = np.where(np.any(masks[i,:,:],
            axis=0))[0]
            # vertical_indicies = np.where(np.any(masks[i,:,:],
            axis=1))[0]
            # if horizontal_indicies.shape[0]:
            #     x1, x2 = horizontal_indicies[[0, -1]]
            #     y1, y2 = vertical_indicies[[0, -1]]
            # else:
            #     x1, x2, y1, y2 = 0, 0, 0, 0
            boxes.append(np.array(bounding_box(masks[i,:,:])))
            # boxes is in the form [num_objects,2,2]
```

```
        # TODO: remove masks and bounding boxes that are too small
        i = 0
        while i<len(obj_ids):
            if (boxes[i][1][0]-boxes[i][0][0]+1)*(boxes[i][1][1]-boxes[i
            ][0][1]+1)<400:
                obj_ids = np.delete(obj_ids, i)
                boxes.pop(i)
                masks = np.delete(masks, i, axis=0)
            else:
                i +=1

        num_objs = len(masks)

        target = {}
        target["boxes"] = boxes
        target["labels"] = obj_ids
        target["masks"] = masks
        target["num_objs"] = num_objs

        if self.transforms is not None:
            img, target = self.transforms(img, target)

        return img, target

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


data = VkittiDataset(full_dataset_path)
img, target = data.__getitem__(2)
```

1.2.2

The transform class CustRandomHorizontalFlip was implemented. The code for detection_utils .py is included Section 3, but the relevant part is included below.

```
# CUSTOM TRANSFORMS

class CustRandomHorizontalFlip(object):
    def __init__(self, prob):
        self.prob = prob

    def __call__(self, image, target):
        # TODO: convert image and target to tensors (ok)
        # F.to_tensor converts input to tensor
        image = F.to_tensor(image)

        target['num_objs'] = torch.tensor([target['num_objs']])
        target['boxes'] =
        torch.from_numpy(np.array(target['boxes']).astype(np.float32))
        target['labels'] = torch.ones((target['num_objs'],),
        dtype=torch.int64)
        target['masks'] =
        torch.from_numpy(target['masks'].astype(np.uint8))
```

```
        ########################################################
        # Previous implementation
        # t = []
        # items = []
        # for item in target:
        #     items.append(item)
        #     if item != 'labels' and item != 'num_objs':
        #         tens = F.to_tensor(np.array(target[item]))
        #     else:
        #         tens = target[item]
        #     t.append(tens)

        # target = []
        # i = 0
        # for item in t:
        #     target.append({items[i]: item})
        #     i += 1
        ########################################################

        # TODO: With a probability of 0.5, flip the image, bounding box
        # and mask horizontally. (ok)
        # Hint: x.flip(axis) flips the tensor x along the provided axis
        p = random.random()
        if p < self.prob:
            image = image.flip(2)
            for i in range(len(target['boxes'])):
                target['masks'][i] =  target['masks'][i].flip(1)
                target['boxes'][i]=torch.tensor(np.array(bounding_box(np
                .array(target['masks'][i]))).flatten())

        return image, target

randFlip = CustRandomHorizontalFlip(0.5)

print(1)
print(target['masks'][0,:])

img, target = randFlip.__call__(img, target)
print(2)
print(target['masks'][0,:])
```

1.2.3

The build_maskrcnn method was completed. The total number of parameters for the model is 60558. The code for detection_utils .py is included in Section 3, but the relevant part is included below.

```
# MODEL

def build_maskrcnn():
    # TODO: build and return a Mask R-CNN model with pretrained
    mobilenet v2 backbone
    backbone = torchvision.models.mobilenet_v2(pretrained =
    True).features
    backbone.out_channels = 1280
    sizes = ((8,16,32,64,128),)
```

```
    aspect_ratios = ((0.5, 0.7, 1.2),)
    anchor_generator = AnchorGenerator(sizes=sizes,
    aspect_ratios=aspect_ratios)
    #anchor_generator.generate_anchors(tuple(sizes),
    tuple(aspect_ratios))

    roi_pooler = torchvision.ops.MultiScaleRoIAlign(featmap_names=['0',
                                output_size=7,
                                sampling_ratio=2)
    model = MaskRCNN(backbone,
                    num_classes=2,
                    rpn_anchor_generator = anchor_generator,
                    box_roi_pool = roi_pooler)

    return model

model = build_maskrcnn()
numOfParams = 0
for parameter in model.parameters():
    numOfParams += len(parameter)
print(numOfParams)
```

1.2.4

The training code is included below. Due to time limitation, although it is checked that the code runs, I did not train the model.

```
# TRAIN

def train(model, train_loader, optimizer, schedule, num_epochs):
    model.train()
    # TODO: Train the Mask R-CNN model

    for e in range(1, num_epochs):
        epoch_loss = 0
        epoch_acc = 0
        for (X_batch, y_batch) in train_loader:
            #print(X_batch)
            #print(y_batch)
            #X_batch, y_batch = X_batch.to(device), y_batch.to(device)

            optimizer.zero_grad()
            loss_dict = model(X_batch, y_batch)

            losses = sum(loss for loss in loss_dict.values())

            losses.backward()
            optimizer.step()
            schedule.step()
            epoch_loss += losses


        print(f'Epoch {e+0:03}: | Loss:
        {epoch_loss/len(train_loader):.5f} | Acc:
        {epoch_acc/len(train_loader):.3f}')
```

```python
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=3,
gamma=0.1)
train(model, train_loader, optimizer, scheduler, 2)
```

1.2.5

The code created for this portion of the assignment is included below, but it is also part of the detection_utils .py file included in Section 3. Due to time limitation, I did not create the required visuals.

```python
# EVALUATE

def eval(model, test_loader):
    model.eval()
    # TODO: Generate visualization of output on a batch of test images
    for X_batch, y_batch in test_loader:
        # TODO: compute the overall accuracy, confusion matrix,
        # precision and recall on the test dataset
        predictions = model(X_batch)
        display_images_wrapper(X_batch[0], predictions)
        break
eval(model, test_loader)
```

2 Problem 2

This week, I spent time with friends drinking coffee in one of the many coffee shops around campus. On Sunday, we also went on a long walk around campus, enjoying the sun and the views.

3 Code Implementation

This code can also be found at `https://github.com/alextzik/navigation_autonomous_systems/tree/main/Deep-Learning`.

3.1 datagen.py

```python
from PIL import Image
import numpy as np
import pandas as pd
import os, sys

# TODO: enter the folder paths to raw data and the output dataset. Set
the Scene variable
scene = "Scene02" # ['Scene01', 'Scene02', 'Scene06', 'Scene18',
'Scene20']
rgb_dataset_path = "/Users/AlexandrosTzikas/Desktop/hw3/vkitti_2.0.3_rgb
/{scene}/clone/frames/{type}/Camera_0/".format(scene=scene, type="rgb")
bbox_path = "/Users/AlexandrosTzikas/Desktop/hw3/vkitti_2.0.3_textgt/{sc
ene}/clone/{type}.txt".format(scene=scene, type="bbox")
class_dataset_path =
"/Users/AlexandrosTzikas/Desktop/hw3/class_dataset/"
```

```
bbox = pd.read_csv(bbox_path, delim_whitespace=True)

def bb_intersection_over_union(boxA, boxB):
    # From: https://github.com/adiprasad/unsup-hard-negative-mining-msco
    co
    # @ "boxA" and "boxB" includes [left, top, right, bottom]

    # determine the (x, y)-coordinates of the intersection rectangle
    xA = max(boxA[0], boxB[0])
    yA = max(boxA[1], boxB[1])
    xB = min(boxA[2], boxB[2])
    yB = min(boxA[3], boxB[3])

    # check if boxes are completed separated.
    if xB-xA < 0 or yB-yA <0:
        return 0

    # compute the area of intersection rectangle
    interArea = (xB - xA + 1) * (yB - yA + 1)

    # compute the area of both the prediction and ground-truth
    rectangles
    boxAArea = (boxA[2] - boxA[0] + 1) * (boxA[3] - boxA[1] + 1)
    boxBArea = (boxB[2] - boxB[0] + 1) * (boxB[3] - boxB[1] + 1)

    # compute the intersection over union by taking the intersection
    # area and dividing it by the sum of prediction + ground-truth
    # areas - the interesection area
    denominator = float(boxAArea + boxBArea - interArea)
    iou = 0
    if denominator == 0: iou = 0
    else: iou = interArea / denominator

    # return the intersection over union value
    return iou

for i in range(len(os.listdir(rgb_dataset_path))):
    img = Image.open(rgb_dataset_path+"rgb_{i}.jpg".format(i=str(i).zfil
    l(5)))
    Iw, Ih = img.size

    pos_crops = []
    neg_boxes = []
    subbox = bbox[(bbox['frame']==i)&(bbox['cameraID']==0)]

    # Gen starting negatives
    for _ in range(len(subbox)):
        boxA = [np.random.randint(0, Iw-100), np.random.randint(0,
        Ih-100), 0, 0]
        wA = np.random.randint(50, 100)
        hA = np.random.randint(50, 100)
        boxA[2] = boxA[0]+wA
        boxA[3] = boxA[1]+hA
        neg_boxes.append(boxA)
```

```
    for _bbox in subbox.iterrows():
        _bbox = _bbox[1]
        boxB = [_bbox['left'], _bbox['top'], _bbox['right'],
        _bbox['bottom']]
        for _boxA in neg_boxes:
            if bb_intersection_over_union(boxA, boxB) > 0.1:
                neg_boxes.remove(_boxA)

        wB = np.abs(_bbox['right']-_bbox['left'])
        hB = np.abs(_bbox['top']-_bbox['bottom'])

        if wB>50 and hB>50:
            crop_box = img.crop(boxB)
            pos_crops.append(crop_box)


    neg_crops = []
    for j in range(min(len(pos_crops), len(neg_boxes))):
        crop_box = img.crop(neg_boxes[j])
        neg_crops.append(crop_box)

    # Save files
    for j, crop in enumerate(pos_crops):
        crop.save(class_dataset_path+"pos/{i}_{j}.jpg".format(i=i, j=j))

    for j, crop in enumerate(neg_crops):
        crop.save(class_dataset_path+"neg/{i}_{j}.jpg".format(i=i, j=j))


######## Calculate mean and standard deviation across channels
sum_R = 0
sum_G = 0
sum_B = 0
numOfPixels = 0

for i in range(len(os.listdir(rgb_dataset_path))):
    img = Image.open(rgb_dataset_path+"rgb_{i}.jpg".format(i=str(i).zfil
    l(5)))
    sum_R += sum(np.asarray(list(img.getdata(0)))/255)
    sum_G += sum(np.asarray(list(img.getdata(1)))/255)
    sum_B += sum(np.asarray(list(img.getdata(2)))/255)
    numOfPixels += img.size[0]*img.size[1]

mean_R = sum_R/numOfPixels
mean_G = sum_G/numOfPixels
mean_B = sum_B/numOfPixels

sum_R = 0
sum_G = 0
sum_B = 0

for i in range(len(os.listdir(rgb_dataset_path))):
    img = Image.open(rgb_dataset_path+"rgb_{i}.jpg".format(i=str(i).zfil
```

```
    l(5)))
    sum_R += sum((np.asarray(list(img.getdata(0)))/255-mean_R)**2)
    sum_G += sum((np.asarray(list(img.getdata(1)))/255-mean_G)**2)
    sum_B += sum((np.asarray(list(img.getdata(2)))/255-mean_B)**2)

sd_R = np.sqrt(sum_R/numOfPixels)
sd_G = np.sqrt(sum_G/numOfPixels)
sd_B = np.sqrt(sum_B/numOfPixels)

print("Mean of R band is", mean_R)
print("St. Dev of R band is", sd_R)

print("Mean of G band is", mean_G)
print("St. Dev of G band is", sd_G)

print("Mean of B band is", mean_B)
print("St. Dev of B band is", sd_B)
```

3.2 classifier_utils .py

```
from numpy.core.numeric import Inf
import torch
from matplotlib import pyplot as plt
import matplotlib.patches as patches
from PIL import Image
import numpy as np
import pandas as pd
import os, sys

from torchvision import datasets
from torchvision import transforms
from torch.utils.data.sampler import SubsetRandomSampler

import torch.nn as nn
import torch.nn.functional as F

device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')
class_dataset_path =
"/Users/AlexandrosTzikas/Desktop/hw3/class_dataset/"

# DATALOADER

def load_data(class_dataset_path):
    batch_size = 8
    num_workers = 0
    test_fraction = 0.1

    transform = transforms.Compose([
        transforms.Resize((32,32)),
        transforms.ToTensor(),
        transforms.Normalize(
            mean = [0.61717, 0.6252, 0.5192],
            std = [0.25209, 0.244, 0.2677]
```

```python
        ),
        transforms.RandomHorizontalFlip(p=0.5)
    ])
    # TODO: Add transforms for preprocessing and data augmentation.
    # You should atleast include:
    # - resize to 32X32 images
    # - Normalize by the mean and std deviation of the dataset
    # - Randomly flip the image horizontally with a probability of 0.5

    dataset = datasets.ImageFolder(class_dataset_path,
    transform=transform)

    num_train = len(dataset)
    indices = list(range(num_train))
    split = int(np.floor(test_fraction * num_train))
    # TODO: Load the classification dataset into train and test loaders
    np.random.shuffle(indices)

    train_idx = indices[split:]
    train_sampler = SubsetRandomSampler(train_idx)

    test_idx = indices[:split]
    test_sampler = SubsetRandomSampler(test_idx)

    print(len(train_idx))
    print(len(test_idx))

    train_loader = torch.utils.data.DataLoader(
        dataset, batch_size = batch_size, sampler = train_sampler,
        num_workers = num_workers
    )
    test_loader = torch.utils.data.DataLoader(
        dataset, batch_size = batch_size, sampler = test_sampler,
        num_workers = num_workers
    )

    return (train_loader, test_loader)

# NEURAL NETWORK ARCHITECTURE

class VehicleClassifier(nn.Module):
    def __init__(self):
        super(VehicleClassifier, self).__init__()
        # TODO: create neural network layers for classification
        self.conv1 = nn.Conv2d(3,6,5)
        self.fc1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(6,16, 5)
        self.fc2 = nn.MaxPool2d(2)
        self.fc3 = nn.Linear(400,120)
        self.fc4 = nn.Linear(120,84)
        self.fc5 = nn.Linear(84,1)
```

```
    def forward(self, x):
        # TODO: write the forward pass of the neural network
        x = F.relu(self.conv1(x))
        #print(x.size())
        x = self.fc1(x)
        #print(x.size())
        x = F.relu(self.conv2(x))
        #print(x.size())
        x = self.fc2(x)
        #print(x.size())
        x = torch.reshape(x, (x.size()[0], x.size()[1]*x.size()[2]*x.size()[3]))
        x = F.relu(self.fc3(x))
        #print(x.size())
        x = F.relu(self.fc4(x))
        #print(x.size())
        x = self.fc5(x)
        #print(x.size())
        x = F.sigmoid(x)

        return x

# Print network structure
net = VehicleClassifier()
print(net)

# LOSS AND METRICS (add more methods here if needed)

def bce_loss(y_pred, y_target):
    # TODO: compute binary cross entropy loss from NN output y_pred and
    target y_target
    bceLoss = nn.BCELoss()
    loss = bceLoss(y_pred, y_target)
    return loss

def binary_acc(y_pred, y_target):
    # TODO: compute accuracy of the NN output y_pred from target
    y_target
    y_acc = y_pred.clone().detach().requires_grad_(False)
    y_acc[y_pred>0.5] = 1
    y_acc[y_pred<=0.5] = 0
    acc = sum(y_acc==y_target)/len(y_pred)
    return acc

def true_pos(y_pred, y_target):
    y_acc = y_pred.clone().detach().requires_grad_(False)
    y_acc[y_pred>0.5] = 1
    y_acc[y_pred<=0.5] = 0
    truePos = sum(y_acc*y_target)
    return truePos

def true_neg(y_pred, y_target):
    y_acc = y_pred.clone().detach().requires_grad_(False)
    y_acc[y_pred>0.5] = 0
    y_acc[y_pred<=0.5] = 1
```

```python
    trueNeg = sum(y_acc*(1-y_target))
    return trueNeg

def false_pos(y_pred, y_target):
    y_acc = y_pred.clone().detach().requires_grad_(False)
    y_acc[y_pred>0.5] = 1
    y_acc[y_pred<=0.5] = 0
    falseNeg = sum(y_acc*(1-y_target))
    return falseNeg

def false_neg(y_pred, y_target):
    y_acc = y_pred.clone().detach().requires_grad_(False)
    y_acc[y_pred>0.5] = 0
    y_acc[y_pred<=0.5] = 1
    falseNeg = sum(y_acc*y_target)
    return falseNeg

# TRAINING

# Optimizer definition
optimizer = torch.optim.Adam(net.parameters(), lr=0.0001)

def train(model, train_loader, optimizer, num_epochs):
    model.train()

    for e in range(1, num_epochs):
        epoch_loss = 0
        epoch_acc = 0
        for X_batch, y_batch in train_loader:
            #(X_batch.shape)
            # TODO: train the model and compute epoch loss and accuracy
            # x.item() returns the number contained within tensor x
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)

            optimizer.zero_grad()
            y_pred_ = model(X_batch)
            y_pred = torch.flatten(y_pred_)

            y_pred = y_pred.to(torch.float32)
            y_batch = y_batch.to(torch.float32)

            loss = bce_loss(y_pred, y_batch)
            acc = binary_acc(y_pred, y_batch)
            loss.backward()
            optimizer.step()
            #break
            epoch_loss += loss
            epoch_acc += acc

        print(f'Epoch {e+0:03}: | Loss:
        {epoch_loss/len(train_loader):.5f} | Acc:
        {epoch_acc/len(train_loader):.3f}')

train_loader, test_loader = load_data(class_dataset_path)
```

```python
train(net, train_loader, optimizer, 21)
# VALIDATION


def test(model, test_loader):
    model.eval()

    with torch.no_grad():
        acc = 0
        truePos = 0
        trueNeg = 0
        falsePos = 0
        falseNeg = 0
        corr_pred = 0
        img_1g = torch.zeros((1,3,32,32))

        for X_batch, y_batch in test_loader:
            # TODO: compute the overall accuracy, confusion matrix,
            # precision and recall on the test dataset
            print(X_batch)
            y_pred_ = model(X_batch)
            y_pred = torch.flatten(y_pred_)

            y_pred = y_pred.to(torch.float32)
            y_batch = y_batch.to(torch.float32)

            truePos += true_pos(y_pred, y_batch)
            trueNeg += true_neg(y_pred, y_batch)
            falseNeg += false_neg(y_pred, y_batch)
            falsePos += false_pos(y_pred, y_batch)
            corr_pred += truePos+trueNeg

            for i in range(len(y_batch)):
                if y_pred[i]>0.5 and y_batch[i]==1:
                    img_1g = X_batch[i,:,:,:]

            #break
        acc = corr_pred/(corr_pred+falseNeg+falsePos)

        if truePos+falsePos > 0:
            precision = truePos/(truePos+falsePos)
        else:
            precision = Inf

        if truePos+falseNeg > 0:
            recall = truePos/(truePos+falseNeg)
        else:
            recall = Inf
        print("Metrics in Test Dataset")
        print("Overall accuracy is:", np.array(acc))
        print("True Positives:", np.array(truePos))
        print("True Negatives:", np.array(trueNeg))
        print("False Negatives:", np.array(falseNeg))
        print("False Positives:", np.array(falsePos))
```

```
        print("Precision:", np.array(precision))
        print("Recall:", np.array(recall))

        images, labels = one_batch(test_loader)
        plot_images_labels(images, labels)
    return img_1g

# VISUALIZATION

# helper function to un-normalize and display an image
def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    plt.imshow(np.transpose(img, (1, 2, 0)))  # convert from Tensor
    image

# obtain one batch of images and labels from data loader
def one_batch(loader):
    dataiter = iter(loader)
    images, labels = dataiter.next()
    images = images.numpy() # convert images to numpy for display
    labels = labels.numpy()
    return images, labels

# display 8 images with labels
def plot_images_labels(images, labels):
    # plot the images in the batch, along with the corresponding labels
    fig = plt.figure(figsize=(10, 4))

    for idx in np.arange(len(images)):
        ax = fig.add_subplot(2, 4, idx+1, xticks=[], yticks=[])
        imshow(images[idx])
        ax.set_title(labels[idx])
    plt.waitforbuttonpress()


###### 1g
img = test(net, test_loader)
img = img[None,:]
print(img.shape)
y_pred_ = net(img)
y_pred = torch.flatten(y_pred_)
print(y_pred)
plot_images_labels(img, y_pred)

noise = transforms.ColorJitter(brightness=10, contrast=20, saturation=0,
hue=0)
img = noise.forward(img)
y_pred_ = net(img)
y_pred = torch.flatten(y_pred_)
print(y_pred)
plot_images_labels(img, y_pred)
```

3.3 detection_utils .py

```
from numpy.core.numeric import Inf
import torch
from matplotlib import pyplot as plt
import matplotlib.patches as patches
from PIL import Image
import numpy as np
import pandas as pd
from matplotlib.collections import PatchCollection
import os, sys
import urllib.request
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
from visualize import *


from torch.utils.data import Dataset
import torchvision

from torchvision import transforms
from torchvision.models.detection.rpn import AnchorGenerator
from torchvision.models.detection import MaskRCNN

from torch.utils.data.sampler import SubsetRandomSampler
import random

from torchvision.transforms import functional as F

# TODO: update the dataset path
full_dataset_path =
"/Users/AlexandrosTzikas/Desktop/hw3/vkitti_2.0.3_rgb/{scene}/clone/fra
mes/{type}/Camera_0/"
device = torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu')

###### Helper function for bounding box
def bounding_box(x):
    pos = np.where(x)
    xmin = np.min(pos[1])
    xmax = np.max(pos[1])
    ymin = np.min(pos[0])
    ymax = np.max(pos[0])
    return (xmin, ymin), (xmax, ymax)

# DATASET CLASS

# Define dataset class
class VkittiDataset(Dataset):
    def __init__(self, full_dataset_path, transforms=None):
        self.root = full_dataset_path
        self.transforms = transforms

        # load all image files, sorting them to
```

```
    # ensure that they are aligned
    scenes = ['Scene01', 'Scene02', 'Scene06', 'Scene18',
    'Scene20']

    self.imgs = []
    for scene in scenes:
        dataset_path = full_dataset_path.format(scene=scene,
        type="rgb")
        for path in os.listdir(dataset_path):
            full_path = os.path.join(dataset_path, path)
            self.imgs.append(full_path)
    self.imgs = list(sorted(self.imgs))

    self.masks = []
    for scene in scenes:
        dataset_path = full_dataset_path.format(scene=scene,
        type="instanceSegmentation")
        for path in os.listdir(dataset_path):
            full_path = os.path.join(dataset_path, path)
            self.masks.append(full_path)
    self.masks = list(sorted(self.masks))

def __getitem__(self, idx):

    img = Image.open(self.imgs[idx]).convert("RGB")    # open image
    and convert to RGB if grayscale
    mask = Image.open(self.masks[idx])   # open mask (no
    conversion)


    mask = np.array(mask)   # convert mask to np array. Mask has
    size (375, 1242) as the image

    #TODO: find all unique entities present in the image. Remove
    any unwanted classes, such as 0 (background) (ok)
    obj_ids = np.sort(np.unique(mask))

    if obj_ids[0]==0:
        obj_ids = np.delete(obj_ids, 0)

    num_objs = len(obj_ids)

    #TODO: for each object id, create a corresponding binary mask
    # Each mask must be of type bool and true where the object is
    present and false everywhere else (ok)
    masks = np.zeros((num_objs, mask.shape[0], mask.shape[1]))
    for i in range(num_objs):
        masks[i,:,:] = (mask==obj_ids[i])


    # get bounding box coordinates from the masks
    boxes = []
    issmall = np.zeros(num_objs, dtype=bool) # 1 if bounding box is
    smaller than 20X20 pixels
```

```
        # TODO: get bounding box coordinates from the masks
        for i in range(num_objs):
            # horizontal_indicies = np.where(np.any(masks[i,:,:],
            axis=0))[0]
            # vertical_indicies = np.where(np.any(masks[i,:,:],
            axis=1))[0]
            # if horizontal_indicies.shape[0]:
            #     x1, x2 = horizontal_indicies[[0, -1]]
            #     y1, y2 = vertical_indicies[[0, -1]]
            # else:
            #     x1, x2, y1, y2 = 0, 0, 0, 0
            boxes.append(np.array(bounding_box(masks[i,:,:])).flatten()
            ) # boxes is in the form [num_objects,2,2]


        # TODO: remove masks and bounding boxes that are too small (ok)
        i = 0
        while i<len(obj_ids):
            if (boxes[i][2]-boxes[i][0]+1)*(boxes[i][3]-boxes[i][1]+1)<
            400:
                obj_ids = np.delete(obj_ids, i)
                boxes.pop(i)
                masks = np.delete(masks, i, axis=0)
            else:
                i +=1

        num_objs = len(masks)

        target = {}
        target["boxes"] = boxes
        target["labels"] = obj_ids
        target["masks"] = masks
        target["num_objs"] = num_objs

        if self.transforms is not None:
            img, target = self.transforms(img, target)

        return img, target

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


data = VkittiDataset(full_dataset_path)
img, target = data.__getitem__(2)


# CUSTOM TRANSFORMS

class CustRandomHorizontalFlip(object):
    def __init__(self, prob):
        self.prob = prob
```

```
    def __call__(self, image, target):
        # TODO: convert image and target to tensors (ok)
        # F.to_tensor converts input to tensor
        image = F.to_tensor(image)

        target['num_objs'] = torch.tensor([target['num_objs']])
        target['boxes'] =
        torch.from_numpy(np.array(target['boxes']).astype(np.float32))
        target['labels'] = torch.ones((target['num_objs'],),
        dtype=torch.int64)
        target['masks'] =
        torch.from_numpy(target['masks'].astype(np.uint8))

        ########################################################
        # Previous implementation
        # t = []
        # items = []
        # for item in target:
        #     items.append(item)
        #     if item != 'labels' and item != 'num_objs':
        #         tens = F.to_tensor(np.array(target[item]))
        #     else:
        #         tens = target[item]
        #     t.append(tens)

        # target = []
        # i = 0
        # for item in t:
        #     target.append({items[i]: item})
        #     i += 1
        ########################################################

        # TODO: With a probability of 0.5, flip the image, bounding box
        # and mask horizontally. (ok)
        # Hint: x.flip(axis) flips the tensor x along the provided axis

        p = random.random()
        if p < self.prob:
            image = image.flip(2)
            for i in range(len(target['boxes'])):
                target['masks'][i] =  target['masks'][i].flip(1)
                target['boxes'][i]=torch.tensor(np.array(bounding_box(n
                p.array(target['masks'][i]))).flatten())

        return image, target

randFlip = CustRandomHorizontalFlip(0.5)

print(1)
print(target['masks'][0,:])

img, target = randFlip.__call__(img, target)
print(2)
print(target['masks'][0,:])
```

```
# DATALOADER

def load_data(full_dataset_path):
    train_batch_size = 8
    test_batch_size = 1
    num_workers = 0
    test_fraction = 0.1

    transform = CustRandomHorizontalFlip(0.5)

    dataset = VkittiDataset(full_dataset_path, transforms=transform)

    # Function to convert batch of outputs to lists instead of tensors
    # in order to support variable sized images
    def collate_fn(batch):
        batch = filter (lambda x:x[1]["num_objs"] > 0, batch)
        return tuple(zip(*batch))

    num_train = len(dataset)
    indices = list(range(num_train))
    split = int(np.floor(test_fraction * num_train))
    # TODO: Load the classification dataset into train and test loaders
    np.random.shuffle(indices)

    train_idx = indices[split:]
    train_sampler = SubsetRandomSampler(train_idx)

    test_idx = indices[:split]
    test_sampler = SubsetRandomSampler(test_idx)

    #print(len(train_idx))
    #print(len(test_idx))
    print(1)
    train_loader = torch.utils.data.DataLoader(
        dataset, batch_size = train_batch_size, sampler =
        train_sampler, num_workers = num_workers, collate_fn =
        collate_fn
    )
    test_loader = torch.utils.data.DataLoader(
        dataset, batch_size = test_batch_size, sampler = test_sampler,
        num_workers = num_workers, collate_fn = collate_fn
    )

    print(1)
    return (train_loader, test_loader)


train_loader, test_loader = load_data(full_dataset_path)


# MODEL
```

```
def build_maskrcnn():
    # TODO: build and return a Mask R-CNN model with pretrained
    mobilenet v2 backbone
    backbone = torchvision.models.mobilenet_v2(pretrained =
    True).features
    backbone.out_channels = 1280
    sizes = ((8,16,32,64,128),)
    aspect_ratios = ((0.5, 0.7, 1.2),)
    anchor_generator = AnchorGenerator(sizes=sizes,
    aspect_ratios=aspect_ratios)

    #anchor_generator.generate_anchors(tuple(sizes),
    tuple(aspect_ratios))

    roi_pooler = torchvision.ops.MultiScaleRoIAlign(featmap_names=['0']
    ,
                                    output_size=7,
                                    sampling_ratio=2)
    model = MaskRCNN(backbone,
                    num_classes=2,
                    rpn_anchor_generator = anchor_generator,
                    box_roi_pool = roi_pooler)

    return model

model = build_maskrcnn()
numOfParams = 0
for parameter in model.parameters():
    numOfParams += len(parameter)
print(numOfParams)


# TRAIN

def train(model, train_loader, optimizer, schedule, num_epochs):
    model.train()
    # TODO: Train the Mask R-CNN model

    for e in range(1, num_epochs):
        epoch_loss = 0
        epoch_acc = 0
        for (X_batch, y_batch) in train_loader:
            #print(X_batch)
            #print(y_batch)
            #X_batch, y_batch = X_batch.to(device), y_batch.to(device)

            optimizer.zero_grad()
            loss_dict = model(X_batch, y_batch)

            losses = sum(loss for loss in loss_dict.values())

            losses.backward()
            optimizer.step()
            schedule.step()
```

```python
            epoch_loss += losses


        print(f'Epoch {e+0:03}: | Loss:
        {epoch_loss/len(train_loader):.5f} | Acc:
        {epoch_acc/len(train_loader):.3f}')



optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=3,
gamma=0.1)
#train(model, train_loader, optimizer, scheduler, 2)


# EVALUATE

def eval(model, test_loader):
    model.eval()
    # TODO: Generate visualization of output on a batch of test images
    for X_batch, y_batch in test_loader:
        # TODO: compute the overall accuracy, confusion matrix,
        # precision and recall on the test dataset
        predictions = model(X_batch)
        display_images_wrapper(X_batch[0], predictions)
        break
eval(model, test_loader)
```