Angel Nieto García 2211798052

Homework #5

1. Source code
2. Dataset: For his class I passed as arguments the root directory and two transformations one for the image we are going to train and a target\_transformation which resizes the output and the ground truth. Image.open was used to open the file as an Image tensor.

class KittiDataset(Dataset): # Pytorch dataset class

# important functions len and getitem

# Load data in an efficient way, like multiple CPUs

# To speed up process in training specially for HD images so it can have multithreading features

"""Face Landmarks dataset."""

def \_\_init\_\_(self, root\_dir, transform=None, target\_transform=None):

"""

Args:

root\_dir (string): Directory with all the images.

"""

self.root\_dir = root\_dir

self.transform = transform

self.target\_transform = target\_transform

self.images = os.listdir(os.path.join(root\_dir, 'image\_2'))

self.labels = os.listdir(os.path.join(root\_dir, 'semantic'))

self.gt = os.listdir(os.path.join(root\_dir, 'semantic\_rgb'))

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

return len(self.images)

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

if torch.is\_tensor(idx):

idx = idx.tolist()

img\_path = os.path.join(self.root\_dir + '/image\_2', self.images[idx])

label\_path = os.path.join(self.root\_dir + '/semantic', self.images[idx])

gt\_path = os.path.join(self.root\_dir + '/semantic\_rgb', self.images[idx])

image = pil\_loader(img\_path)

label = Image.open(label\_path)

gt = Image.open(gt\_path)

sample = {'image': image, 'label': label, 'gt': gt}

if self.transform:

sample['image'] = self.transform(sample['image'])

sample['label'] = self.target\_transform(sample['label'])

sample['gt'] = self.target\_transform(sample['gt'])

return sample

1. Models: Uses additional Conv layers to get better classification and at the end we add a fully connected layer with Dropouts to increase variance in the distribution.

FCN 32: Use as described in homework with a slight modification of filter size of 224 in the last layer, since I was not able to make it work with filter\_size of 64 or 32, probably I need to add some padding in the previous layer to make it work with such filter size.

num\_classes = 35

class FCN32(nn.Module):

def \_\_init\_\_(self):

super(FCN32, self).\_\_init\_\_()

self.features = vgg16.features

self.classifier = nn.Sequential(

nn.Conv2d(512, 4096, 7),

nn.ReLU(inplace=True),

nn.Dropout2d(),

nn.Conv2d(4096, 4096, 1),

nn.ReLU(inplace=True),

nn.Dropout2d(),

nn.Conv2d(4096, num\_classes, 1),

nn.ConvTranspose2d(num\_classes, num\_classes, 224, stride=32)

)

def forward(self, x):

x = self.features(x)

x = self.classifier(x)

return x

fcn = FCN32()

fcn.to(device)

FCN 16: Implemented as described in the homework

num\_classes = 35

class FCN16(nn.Module):

def \_\_init\_\_(self):

super(FCN16, self).\_\_init\_\_()

self.features = vgg16.features

self.classifier = nn.Sequential(

nn.Conv2d(512, 4096, 7),

nn.ReLU(inplace=True),

nn.Conv2d(4096, 4096, 1),

nn.ReLU(inplace=True),

nn.Conv2d(4096, num\_classes, 1)

)

self.score\_pool4 = nn.Conv2d(512, num\_classes, 1)

self.upscore2 = nn.ConvTranspose2d(num\_classes, num\_classes, 14, stride=2, bias=False)

self.upscore16 = nn.ConvTranspose2d(num\_classes, num\_classes, 16, stride=16, bias=False)

def forward(self, x):

pool4 = self.features[:-7](x)

pool5 = self.features[-7:](pool4)

pool5\_upscored = self.upscore2(self.classifier(pool5))

pool4\_scored = self.score\_pool4(pool4)

combined = pool4\_scored + pool5\_upscored

res = self.upscore16(combined)

return res

fcn = FCN16()

fcn.to(device)

c) Loss Function and Optimizer

criterion = nn.CrossEntropyLoss()

betas = (0.5, 0.999)

optimizer = optim.Adam(fcn.parameters(), lr=0.001, betas=betas)

d) Optimizer

optimizer = optim.Adam(net.parameters(), lr=0.001)

e) Evaluation: To calculate mean IoU I used the confusion matrix data which pixel-level IoU easy to calculate since the intersection is calculated using the diagonal of the confusion matrix (TP) and the union (TP+PN+FN) is calculated by the summation in axis 1 + the summation in axis 0 – intersection.

overall\_conf\_mat = np.zeros((35, 35))

with torch.no\_grad():

for k, dat1 in enumerate(test\_loader):

# Get image, label pair

inputs1, labels1 = dat1['image'], dat1['label']

# Using GPU

inputs1 = inputs1.to(device)

labels1 = labels1.to(device)

# Predicting segmentation for val inputs

outputs1 = model(inputs1)

preds = torch.argmax(outputs1, dim=1).detach().cpu().numpy()

gt = labels1.detach().cpu().numpy()

# Compute confusion matrix

conf\_mat = confusion\_matrix(y\_pred=preds.flatten(), y\_true=gt.flatten(), labels=list(range(35)))

overall\_conf\_mat += conf\_mat

mean\_iou, iou = get\_mean\_iou(conf\_mat=overall\_conf\_mat)

print('IOU: {}'.format(iou))

print('Mean IOU: {}'.format(np.round(mean\_iou, 2)))

with torch.no\_grad():

for k, dat in enumerate(test\_loader):

# Get image, label pair

inputs, labels, gt = dat['image'], dat['label'], dat['gt']

# Using GPU

inputs = inputs.to(device)

labels = labels.to(device)

outputs = model(inputs)

pred = torch.argmax(outputs.squeeze(), dim=0).detach().cpu().numpy()

# Getting the segmentation

segmentation = decode\_segmap(pred)

unorm = UnNormalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225))

unnormalized\_image = unorm(inputs[0].cpu())

image\_array = np.transpose(unnormalized\_image.numpy(), (1, 2, 0))

# gt\_array = np.transpose(gt, (1, 2, 0))

plt.imshow(image\_array); plt.axis('off');

plt.show()

plt.imshow(segmentation); plt.axis('off');

plt.show()

plt.imshow(gt[0]); plt.axis('off');

plt.show()

if k == 20:

break

f) Training Loop: It is split in two parts: the first part for the training where it calculates the loss and IoU in each iteration and once the epoch is completed it calculates the average IoU and average loss, the second part iterates over the validation loader, calculates average IoU and loss, and stores the best model to use it in the evaluation with the test set.

best\_loss = 1000000000

num\_epochs = 140

train\_loss\_history = []

val\_loss\_history = []

train\_iou\_history = []

val\_iou\_history = []

for epoch in range(num\_epochs):

val\_running\_iou = 0

val\_running\_loss = 0

train\_running\_iou = 0

train\_running\_loss = 0

j = 0

fcn.train()

for i, dat in enumerate(train\_loader):

j += 1

# Get image, label pair

inputs, labels = dat['image'], dat['label']

# Using GPU

inputs = inputs.to(device)

labels = labels.to(device)

# Set parameter gradients to 0

optimizer.zero\_grad()

# Forward pass for a batch

outputs = fcn(inputs)

preds = torch.argmax(outputs, dim=1).detach().cpu().numpy()

gt = labels.detach().cpu().numpy()

# Compute loss

loss = criterion(outputs, labels)

train\_running\_loss += loss

# Compute confusion matrix

conf\_mat = confusion\_matrix(y\_pred=preds.flatten(), y\_true=gt.flatten(), labels=list(range(35)))

mean\_iou = get\_mean\_iou(conf\_mat=conf\_mat)

train\_running\_iou += mean\_iou

# Backpropagate

loss.backward()

# Update the weights

optimizer.step()

# Averaging loss and scores

avg\_train\_loss = float(train\_running\_loss)/(j)

avg\_train\_iou = float(train\_running\_iou)/(j)

train\_loss\_history.append(avg\_train\_loss)

train\_iou\_history.append(avg\_train\_iou)

fcn.eval()

with torch.no\_grad():

for k, dat1 in enumerate(val\_loader):

# Get image, label pair

inputs1, labels1 = dat1['image'], dat1['label']

# # Using GPU

inputs1 = inputs1.to(device)

labels1 = labels1.to(device)

# Predicting segmentation for val inputs

outputs1 = fcn(inputs1)

# Compute CE loss and aggregate it

loss1 = criterion(outputs1, labels1)

val\_running\_loss += loss1

# Reshaping prediction segmentations and actual segmentations for iou and dice score

preds = torch.argmax(outputs1, dim=1).detach().cpu().numpy()

gt = labels1.detach().cpu().numpy()

# Compute confusion matrix

conf\_mat = confusion\_matrix(y\_pred=preds.flatten(), y\_true=gt.flatten(), labels=list(range(35)))

# Computing iou and dice scores and aggregating them

mean\_iou = get\_mean\_iou(conf\_mat=conf\_mat)

val\_running\_iou += mean\_iou

avg\_val\_loss = float(val\_running\_loss)/(k+1)

avg\_val\_iou = float(val\_running\_iou)/(k+1)

val\_loss\_history.append(avg\_val\_loss)

val\_iou\_history.append(avg\_val\_iou)

if avg\_val\_loss < best\_loss:

best\_loss = avg\_val\_loss

torch.save(fcn.state\_dict(), '/content/best\_model\_fcn16.pth.tar')

# Visualizations for batch wise metrics

print('epoch {}, training loss: {}, iou score: {}'.format(epoch+1, avg\_train\_loss, avg\_train\_iou))

print('epoch {}, validation loss: {}, iou score: {}'.format(epoch+1, avg\_val\_loss, avg\_val\_iou))

print('Finished Training')

Please see appendix for rest of utility functions, transformations and more.

1. Discussion

2.1 Evolution of training losses and validation losses with FCN 16 using Adam as optimizer, learning rate = 0.001, betas = (0.5, 0.999) after 140 epochs

Chart

Description automatically generated

Chart, scatter chart

Description automatically generated

2.1.1 Evaluation metric on test data

Mean IOU: 0.3

Pixel-level IOU: [0. 0. 0. 0. 0.421875 0.05075758

0.2601626 0.87887102 0.55460552 0.31881262 0.37165304 0.7448051

0.19798761 0.09071496 0.4253857 0.00647715 0.00618673 0.11895879

0.03682688 0.13758515 0.27939644 0.79163026 0.61728869 0.79067183

0.00425894 0.00504881 0.76342157 0.05257423 0.69709763 0.20661157

0.57954545 0.276 0.0083682 0.373297 nan]

\*nan values seem related to category -1

A picture containing diagram

Description automatically generatedA close up of a street

Description automatically generatedA picture containing graphical user interface

Description automatically generated

A view of a city street

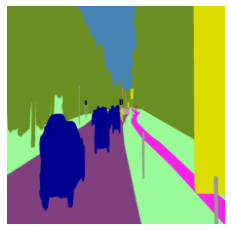
Description automatically generatedA picture containing building, clock

Description automatically generatedA picture containing clock

Description automatically generated

A view of a road

Description automatically generatedA picture containing chart

Description automatically generated

Examples: left original image, center predicted image, right ground truth

2.1.2 Evolution of training losses and validation losses with FCN 32 using Adam as optimizer, learning rate = 0.001, betas = (0.5, 0.999) after 140 epochs

A picture containing chart

Description automatically generated

A picture containing graphical user interface

Description automatically generated

2.1.3 Evaluation metric on test data FCN 32

Mean IOU: 0.12

Pixel-level IOU: [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00

2.68901912e-01 5.55144338e-02 3.84732206e-03 6.77009575e-01

1.54676162e-01 9.13115246e-02 1.29398210e-02 1.88490698e-01

8.81080202e-02 1.87995994e-02 1.81493183e-01 1.15283267e-02

8.59950860e-03 1.13996254e-01 4.69784768e-02 9.04286128e-02

3.19874779e-02 5.70181738e-01 4.17274856e-01 4.85156984e-01

3.56576862e-03 4.88102502e-03 2.87682197e-01 0.00000000e+00

5.64334086e-04 0.00000000e+00 9.09090909e-03 2.18340611e-01

0.00000000e+00 0.00000000e+00 nan]

Examples of FCN 32

A picture containing scene, road, outdoor, train

Description automatically generatedChart

Description automatically generatedA picture containing diagram

Description automatically generated

A close up of a busy city street

Description automatically generatedMap

Description automatically generatedA picture containing object, clock, monitor, screen

Description automatically generated

A view of a city street

Description automatically generatedA picture containing chart

Description automatically generatedA picture containing object, clock

Description automatically generated

As it can be appreciated FCN 32 failed to recognize small and medium size object. I believe this is due to the large filter size at the end of the classifier layers.

* 1. Effects of parameter choices

The most important parameters to have a good semantic segmentation network are without a doubt the optimizer and apply the correct transformations and normalization. Rest of parameters were applied as suggested in homework instructions.

params = {'batch\_size': 5,

'shuffle': True,

'num\_workers': 4}

data\_mean = [0.485, 0.456, 0.406]

data\_std = [0.229, 0.224, 0.225]

The following transformations were applied to the three data sets:

img\_transform = transforms.Compose([

transforms.Resize(input\_size),

transforms.ToTensor(),

transforms.Normalize(mean=data\_mean, std=data\_std)

]) # Applied to input

target\_transform = transforms.Compose([

transforms.Resize(input\_size),

ConvertToBackground()

]) # Applied to output and ground\_truth

* + 1. SGD Optimizer with learning rate of 0.001, momentum 0.99 and 140 epochs

As it can be appreciated in the following images a different optimizer such as SGD produces worse results than using Adam (lower mIoU and higher loss)

A picture containing shape

Description automatically generated

A picture containing shape

Description automatically generated

Test set:

Mean IOU: 0.13

Pixel level-IOU: [0. 0. 0. 0. 0.00673905 0.

0. 0.77584447 0.24534817 0.00775155 0.05853094 0.35657824

0.07689936 0.00583242 0.18200398 0. 0. 0.01159069

0. 0.02248423 0.01655507 0.72896022 0.50053674 0.80410127

0. 0. 0.54987353 0. 0. 0.

0. 0. 0. 0. nan]

A car on a city street

Description automatically generatedA picture containing clock, meter

Description automatically generatedA picture containing clock, monitor, screen, computer

Description automatically generated

A car parked on the side of a road

Description automatically generatedMap

Description automatically generatedA picture containing clock

Description automatically generated

Training classifier parameters only. Adam optimizer, learning rate = 0.001, betas = (0.5, 0.999) after 140 epochs

A picture containing text

Description automatically generated

A picture containing text

Description automatically generated

Test set:

Mean IOU: 0.11

Pixel-level IOU: [0. 0. 0. 0. 0.28793388 0.05206349

0.01512478 0.66271618 0.22499368 0.0575155 0.0733171 0.18720653

0.04215263 0.0217372 0.15904752 0.01124663 0.00980232 0.06606124

0.04008252 0.09322671 0.04233366 0.54269537 0.42301755 0.50308682

0.00745474 0.00648618 0.22163123 0.00782998 0.11574279 0.00813008

0.00381679 0.00429185 0. 0.00275482 nan]

A view of a city street

Description automatically generatedA picture containing building, colorful

Description automatically generatedA picture containing clock

Description automatically generated

A view of the side of a road

Description automatically generatedA picture containing shape

Description automatically generatedGraphical user interface

Description automatically generated

Training only vgg16.features.parameters seems to produce good results in some object like roads, sidewalks, light bulbs, trees and sky, but it performs poorly classifying cars and trucks.

* 1. Example of failed cases

A picture containing scene, road, outdoor, train

Description automatically generatedA picture containing map

Description automatically generatedA picture containing graphical user interface

Description automatically generated

A path with trees on the side of a road

Description automatically generatedMap

Description automatically generatedA picture containing monitor, sign, screen, person

Description automatically generated

It is worth mentioning that in circumstances where there is plenty of shadow even the model with highest average IoU fails to classify the object under the right category. In the first example it was not able to segment the wall and in the second everything many things in the right side were predicted incorrectly. Additionally, some thin object also fails to segment like the protection between the opposite sides of the road in the second example.

1. Conclusions

In general, a customized VGG16 network produces acceptable results for semantic segmentation, but a proper data preprocessing and tunning of hyper parameters is necessary to obtain good results such as freezing/unfreezing convolution layers, pretraining or not VGG16, transformations, normalization, batch size, to name a few. In this experiment it is hard to tell under which circumstances FCN 16 is preferable over FCN 32 and the other way around since in all my experiments FCN 16 performed better than FCN 32 but most probably it was because of an improper configuration of FCN 32 which I believe could be fixed by using some padding in the convolutional layers or adding intermediate layers between the classifier and the deconvolutional layer.

It is also worth mentioning that in both FCN 16 and FCN 32 with Adam optimizer reach their optimal capacity around 20 epochs and then the generalization gap starts increasing so it might be worth to do more fine tuning and try to apply other transformations to the output to try to reduce overfitting but for matters of time it was not possible to do more experiments.

Appendix:

Code to use Google Drive in Colab instead of downloading data every time

from google.colab import drive

ROOT = "/content/drive"

drive.mount(ROOT)

Utility function to transform image array into tensor

class ConvertToBackground(object):

def \_\_call\_\_(self, img):

img = np.asarray(img, dtype=np.long)

img[img == 255] = 0

img = torch.from\_numpy(img)

return img

Utility function to unnormalize input to display it along output and ground truth

class UnNormalize(object):

def \_\_init\_\_(self, mean, std):

self.mean = mean

self.std = std

def \_\_call\_\_(self, tensor):

"""

Args:

tensor (Tensor): Tensor image of size (C, H, W) to be normalized.

Returns:

Tensor: Normalized image.

"""

for t, m, s in zip(tensor, self.mean, self.std):

t.mul\_(s).add\_(m)

# The normalize code -> t.sub\_(m).div\_(s)

return tensor

Utility function to assign values to output image based in predictions

def decode\_segmap(image, nc=35):

label\_colors = np.array([(0, 0, 0),

(0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (111, 74, 0),

( 81, 0, 81), (128, 64,128), (244, 35,232), (250,170,160), (230,150,140),

( 70, 70, 70), (102,102,156), (190,153,153), (180,165,180), (150,100,100),

(150,120, 90), (153,153,153), (153,153,153), (250,170, 30), (220,220,0),

(107,142, 35), (152,251,152), (70,130,180), (220, 20, 60), (255, 0, 0),

( 0, 0,142), ( 0, 0, 70), ( 0, 60,100), ( 0, 0, 90), ( 0, 0,110),

( 0, 80,100), ( 0, 0,230), (119, 11, 32), ( 0, 0,142)])

r = np.zeros\_like(image).astype(np.uint8)

g = np.zeros\_like(image).astype(np.uint8)

b = np.zeros\_like(image).astype(np.uint8)

for l in range(0, nc):

idx = image == l

r[idx] = label\_colors[l, 0]

g[idx] = label\_colors[l, 1]

b[idx] = label\_colors[l, 2]

rgb = np.stack([r, g, b], axis=2)

return rgb

Code to download pretrained vgg16 model

vgg16 = models.vgg16(pretrained=True)

Code to freeze convolution layers

for param in vgg16.features.parameters():

param.requires\_grad = False

Utility function to calculate Pixel level IoU and Mean IoU taking as input a confusion matrix

def get\_mean\_iou(conf\_mat, multiplier=1.0):

cm = conf\_mat.copy()

np.fill\_diagonal(cm, np.diag(cm) \* multiplier)

inter = np.diag(cm)

gt\_set = cm.sum(axis=1)

pred\_set = cm.sum(axis=0)

union\_set = gt\_set + pred\_set - inter

iou = inter.astype(float) / union\_set

mean\_iou = np.nanmean(iou)

return mean\_iou, iou

Code to visualize input, output and ground truth

with torch.no\_grad():

for k, dat in enumerate(test\_loader):

# Get image, label pair

inputs, labels, gt = dat['image'], dat['label'], dat['gt']

# Using GPU

inputs = inputs.to(device)

labels = labels.to(device)

outputs = model(inputs)

pred = torch.argmax(outputs.squeeze(), dim=0).detach().cpu().numpy()

# Getting the segmentation

segmentation = decode\_segmap(pred)

unorm = UnNormalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225))

unnormalized\_image = unorm(inputs[0].cpu())

image\_array = np.transpose(unnormalized\_image.numpy(), (1, 2, 0))

# gt\_array = np.transpose(gt, (1, 2, 0))

plt.imshow(image\_array); plt.axis('off');

plt.show()

plt.imshow(segmentation); plt.axis('off');

plt.show()

plt.imshow(gt[0]); plt.axis('off');

plt.show()

if k == 30:

break