


finding the bounding
box



Object Detection and Localization with Deep Networks

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Tuesday 3rd March, 2020 11:05

Preamble

Most people would say that object detection in images is a more difficult problem than image classification.

Object detection is made challenging by the fact that a good solution to this problem must also do a good job of localizing the object. And when an image contains multiple objects of interest, an object detector must identify them and localize them individually.

Object localization consists, at the least, of estimating the bounding-box rectangle for the object.

Outline

- 1 PurdueShapes5 — A Dataset of Small-Sized Images to Experiment with Object Detection
- 2 A Custom Dataloader for PurdueShapes5
- 3 Detecting and Localizing Objects
- 4 Learning from the Dual Outputs from the Network

Outline

- 1 **PurdueShapes5 — A Dataset of Small-Sized Images to Experiment with Object Detection**
- 2 A Custom Dataloader for PurdueShapes5
- 3 Detecting and Localizing Objects
- 4 Learning from the Dual Outputs from the Network

The PurdueShapes5 Dataset with Bbox Annotations

- I have been impressed with how useful the CIFAR-10 dataset has become for demonstrating in a classroom setting several of the core notions related to image classification with deep networks.
- I felt that there was a need for a similar dataset based on small images (just 32×32) (or, perhaps, 64×64 in the future) for demonstrating concepts related to object detection and the regression needed for estimating the bounding boxes for the detected objects.
- So I have created the PurdueShapes5 dataset to fill this void. [It is usable but still work in progress.]
- The program that generates the dataset also generates the bounding-box (bbox) annotations for the objects. You need the bbox annotations for solving the object localization problem with a neural network.

Some Example Images from the PurdueShapes5 Dataset



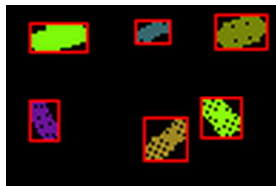
(a) random stars



(b) with bbox annotations



(a) noisy ovals



(b) with bbox annotations

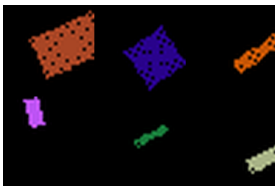
Some Example Images from the PurdueShapes5 Dataset (contd.)



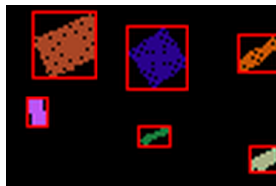
(a) random triangles



(b) with bbox annotations



(a) noisy rectangles

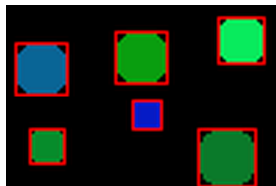


(b) with bbox annotations

Some Example Images from the PurdueShapes5 Dataset (contd.)



(a) random disks



(b) with bbox annotations

- This dataset is available in the following files in the “data” subdirectory of the DLStudio distribution (version 1.0.7). You will see the following archive files there:
 - PurdueShapes5-10000-train.gz
 - PurdueShapes5-1000-test.gz
 - PurdueShapes5-20-train.gz
 - PurdueShapes5-20-test.gz

Data Format Used for the PurdueShapes5 Dataset

- Each 32×32 image is stored in the dataset as the following format:
vspace0.1in

Image stored as the list:

[R, G, B, Bbox, Label]

where

R : is a 1024 element list of int values for the red component of the color at all the pixels

B : the same as above but for the blue component of the color

G : the same as above but for the green component of the color

Bbox : a list like [x1,y1,x2,y2] that defines the bounding box for the object in the image

Label : the shape category of the object

- I serialize the dataset with Python's `pickle` module and then compress it with Python's `gzip` module.

Representing the Images in PurdueShapes5?

- The PIL's Image class has a convenient function `getdata()` that returns in a single call all the pixels in an image as a list of 3-element tuples:

```
data = list(im.getdata())
R = [pixel[0] for pixel in data]          ## data for the input channels
G = [pixel[1] for pixel in data]
B = [pixel[2] for pixel in data]

## Find bounding rectangle
non_zero_pixels = []
for k,pixel in enumerate(data):
    x = k % 32
    y = k // 32
    if any( pixel[p] is not 0 for p in range(3) ):
        non_zero_pixels.append((x,y))
min_x = min( [pixel[0] for pixel in non_zero_pixels] )
max_x = max( [pixel[0] for pixel in non_zero_pixels] )
min_y = min( [pixel[1] for pixel in non_zero_pixels] )
max_y = max( [pixel[1] for pixel in non_zero_pixels] )
```

- Subsequently, you can call on Python's `pickle` to serialize the data for its persistent storage:

```
dataset,label_map = gen_dataset(how_many_images)
serialized = pickle.dumps([dataset, label_map])
f = gzip.open(dataset_name, 'wb')
f.write(serialized)
```

Outline

- 1 PurdueShapes5 — A Dataset of Small-Sized Images to Experiment with Object Detection
- 2 A Custom Dataloader for PurdueShapes5**
- 3 Detecting and Localizing Objects
- 4 Learning from the Dual Outputs from the Network

Custom Data Loaders and PyTorch

- Creating a custom data loader for a DL framework is not as simple as what you did for your second homework? All you had to there was to extend the `torchvision.datasets.CIFAR10` class and tell it that you only wanted to download data for the two images classes, cat and dog.
- Creating a custom data loader is a bit more complex when your data storage format is nothing like used in any of the standard datasets that Torchvision knows about.
- The new inner class `CustomDataLoading` is an attempt at creating a data loader from the ground up. To understand what I mean by that, let's first look at the data format used in the `PurdueShapes5` dataset.
- The dataset was generated by randomizing the sizes and the orientations of these five patterns. Since the patterns are rotated with a very simple non-interpolating transform, just the act of random rotations can introduce boundary and even interior noise in the patterns

A Custom Dataloader for PurdueShapes5

- You must extend the class `torch.utils.data.Dataset` and provide your own implementations for the methods `__len__()` and `__getitem__()`:

```
class PurdueShapes5Dataset(torch.utils.data.Dataset):
    def __init__(self, dl_studio, dataset_file, transform=None):
        super(DLStudio.CustomDataLoading.PurdueShapes5Dataset, self).__init__()
        root_dir = dl_studio.dataroot
        f = gzip.open(root_dir + dataset_file, 'rb')
        dataset = f.read()
        self.dataset, self.label_map = pickle.loads(dataset)
        # reverse the key-value pairs in the label dictionary:
        self.class_labels = dict(map(reversed, self.label_map.items()))
        self.transform = transform

    def __len__(self):
        ## must return the size of the object
        return len(self.dataset)

    def __getitem__(self, idx):
        ## extracts each image, etc., from dataset
        r = np.array( self.dataset[idx][0] )
        g = np.array( self.dataset[idx][1] )
        b = np.array( self.dataset[idx][2] )
        R,G,B = r.reshape(32,32), g.reshape(32,32), b.reshape(32,32)
        im_tensor = torch.zeros(3,32,32, dtype=torch.float)
        im_tensor[0,:,:] = torch.from_numpy(R)
        im_tensor[1,:,:] = torch.from_numpy(G)
        im_tensor[2,:,:] = torch.from_numpy(B)
        sample = {'image' : im_tensor,
                  'bbox' : self.dataset[idx][3],
                  'label' : self.dataset[idx][4] }
        return sample

def load_PurdueShapes5_dataset(self, dataset_server_train, dataset_server_test ):
    transform = tvf.Compose([tvf.ToTensor(),
                             tvf.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
    self.train_data_loader = torch.utils.data.DataLoader(dataset_server_train,
                                                           batch_size=self.dl_studio.batch_size,shuffle=True, num_workers=0)
    self.test_data_loader = torch.utils.data.DataLoader(dataset_server_test,
                                                         batch_size=self.dl_studio.batch_size,shuffle=False, num_workers=0)
```

IMPORTANT: PurdueShapes5 is Still a Work-in-Progress

- The dataloader presented in the inner class of does work but it is not yet a finished product — in the sense that it does not yet have multi-worker support.
- Also, at the moment, it uncompresses the archived data whenever it is called afresh.
- A faster implementation would uncompress it the first and then store the uncompressed version locally in a disk-based hash table of some sort so that any future downloads from the dataset would be much faster. I hope to do that in a future version of DLStudio.

Outline

- 1 PurdueShapes5 — A Dataset of Small-Sized Images to Experiment with Object Detection
- 2 A Custom Dataloader for PurdueShapes5
- 3 Detecting and Localizing Objects**
- 4 Learning from the Dual Outputs from the Network

Difference Between Classifying an Image and Detecting Objects Therein

- You already know how to create a deep network for classifying an entire image. [NOTE: This statement could be extended to say that the same logic *could* be used for classifying the pixel blobs that you get by scanning an image with a window. While logically possible, now you can use encoder-decoder network like the Unet for doing the same.]
- Classifying an image (or a blob) is a single goal that is easily fulfilled in a neural network by looking at the maximum value (or some variant thereof) in the final output.
- But how do you achieve two goals simultaneously: detecting and localizing?
- That is, how do you get a neural network to simultaneously output the label for a blob of pixels and also provide a bounding box for the pixels that correspond to that label?

Creating Multiple Pathways in the Forward Dataflow in a Network

- The main idea here is to create multiple parallel pathways in a neural network that may share some of the initial layers.
- With parallel pathways, for the same input data, you can use a different target for each output, creating multiple losses.
- Subsequently, you would backpropagate with respect to each loss and calculate the gradients of the learnable parameters as you go along.
- That raises the question: Doesn't that mess up the learnable parameters that are shared by the pathways?
- Estimating the bounding box (through either the coordinates of its corners or through a mask) is referred to as solving **the bbox regression problem**.

The LOADnet (for Localizing And Detecting) Classes in DLStudio

- The inner class DetectAndLocalize contains a couple of different versions of the LOADnet network for experimenting with different topologies for predicting both the object class and its bounding box.
- One can argue whether one needs as much convolutional depth in the bbox regression part of a network as in the labeling part.
- The labeling part needs convolutional depth because you do not know in advance at what level of data abstraction the objects in the image would be best detectable.
- For the regression part, if you are directly predicting the corners, perhaps being at the same abstraction as for the labeling part is not that important.

The LOADnet1 Network

```

class LOADnet1(nn.Module):
    def __init__(self, skip_connections=True, depth=32):
        super(DLStudio.DetectAndLocalize.LOADnet1, self).__init__()
        self.pool_count = 3
        self.depth = depth // 2
        self.conv = nn.Conv2d(3, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.skip64 = DLStudio.SkipConnections.SkipBlock(64, 64, skip_connections=skip_connections)
        ...
        self.fc1 = nn.Linear(128 * (32 // 2**self.pool_count)**2, 1000)
        self.fc2 = nn.Linear(1000, 5)
        self.fc3 = nn.Linear(32768, 1000)
        self.fc4 = nn.Linear(1000, 4)

    def forward(self, x):
        x = self.pool(torch.nn.functional.relu(self.conv(x)))
        ## The labeling section:
        for _ in range(self.depth // 4):
            x1 = self.skip64(x)
            x1 = self.skip64ds(x1)
        for _ in range(self.depth // 4):
            x1 = self.skip64(x1)
            x1 = self.skip64to128(x1)
        for _ in range(self.depth // 4):
            x1 = self.skip128(x1)
            x1 = self.skip128ds(x1)
        for _ in range(self.depth // 4):
            x1 = self.skip128(x1)
            x1 = x1.view(-1, 128 * (32 // 2**self.pool_count)**2 )
            x1 = torch.nn.functional.relu(self.fc1(x1))
            x1 = self.fc2(x1)

        ## Only the FC layers used for the Bounding Box regression:
        x2 = x.view(-1, 32768 )
        x2 = torch.nn.functional.relu(self.fc3(x2))
        x2 = self.fc4(x2)
        return x1,x2

```

The LOADnet2 Network

```

class LOADnet2(nn.Module):
    def __init__(self, skip_connections=True, depth=32):
        super(DLStudio.DetectAndLocalize.LOADnet2, self).__init__()
        self.pool_count = 3
        self.depth = depth // 2
        self.conv = nn.Conv2d(3, 64, 3, padding=1)
        ...

    def forward(self, x):
        x = self.pool(torch.nn.functional.relu(self.conv(x)))
        ## The labeling section:
        for _ in range(self.depth // 4):
            x1 = self.skip64(x)
            x1 = self.skip64ds(x1)
        for _ in range(self.depth // 4):
            x1 = self.skip64(x1)
            x1 = self.skip64to128(x1)
        for _ in range(self.depth // 4):
            x1 = self.skip128(x1)
            x1 = self.skip128ds(x1)
        for _ in range(self.depth // 4):
            x1 = self.skip128(x1)
            x1 = x1.view(-1, 128 * (32 // 2**self.pool_count)**2 )
            x1 = torch.nn.functional.relu(self.fc1(x1))
            x1 = self.fc2(x1)

        ## The Bounding Box regression:
        for _ in range(self.depth // 4):
            x2 = self.skip64(x)
            x2 = self.skip64ds(x2)
        for _ in range(self.depth // 4):
            x2 = self.skip64(x2)
            x2 = self.skip64to128(x2)
        for _ in range(self.depth // 4):
            x2 = self.skip128(x2)
            x2 = self.skip128ds(x2)
        for _ in range(self.depth // 4):
            x2 = self.skip128(x2)
            x2 = x2.view(-1, 128 * (32 // 2**self.pool_count)**2 )
            x2 = torch.nn.functional.relu(self.fc3(x2))
            x2 = self.fc4(x2)
        return x1, x2

```

Outline

- 1 PurdueShapes5 — A Dataset of Small-Sized Images to Experiment with Object Detection
- 2 A Custom Dataloader for PurdueShapes5
- 3 Detecting and Localizing Objects
- 4 Learning from the Dual Outputs from the Network**

Learning from the Labeling Loss and the Regression Loss

```
def run_code_for_training(self, net):
    ...
    criterion1 = nn.CrossEntropyLoss()
#   criterion2 = self.dl_studio.DetectAndLocalize.IOULoss(self.dl_studio.batch_size)
    criterion2 = nn.BCELoss()
    optimizer = optim.SGD(net.parameters(),
                           lr=self.dl_studio.learning_rate, momentum=self.dl_studio.momentum)

    for epoch in range(self.dl_studio.epochs):
        running_loss_labeling = 0.0
        running_loss_regression = 0.0
        for i, data in enumerate(self.train_dataloader):
            ...
            gt_too_small = False
            inputs, bounding_box, labels = data['image'], data['bbox'], data['label']
            inputs = inputs.to(self.dl_studio.device)
            labels = labels.to(self.dl_studio.device)
            bounding_box = bounding_box.to(self.dl_studio.device)

            optimizer.zero_grad()

            outputs = net(inputs)
            outputs_label = outputs[0]
            outputs_regression = outputs[1]

            loss_labeling = criterion1(outputs_label, labels)

            mask_regress = torch.zeros(self.dl_studio.batch_size, 32, 32, requires_grad=False)
            mask_gt = torch.zeros(self.dl_studio.batch_size, 32, 32)
            for k, out_regress in enumerate(outputs_regression):
                x1, y1, x2, y2 = out_regress[k].tolist()
                x1_gt, y1_gt, x2_gt, y2_gt = bounding_box[k].tolist()
                x1, y1, x2, y2 = [int(item) if item > 0 else 0 for item in (x1, y1, x2, y2)]
                x1_gt, y1_gt, x2_gt, y2_gt = [int(item) if item > 0 else 0 for item in (x1_gt, y1_gt, x2_gt, y2_gt)]
                if abs(x1_gt - x2_gt) < 5 or abs(y1_gt - y2_gt) < 5: gt_too_small = True
                mask_regress_np = np.zeros((32, 32), dtype=bool)
                mask_gt_np = np.zeros((32, 32), dtype=bool)
                mask_regress_np[y1:y2, x1:x2] = 1
```

Learning from the Labeling Loss and the Regression Loss (contd.)

```
mask_gt_np[y1_gt:y2_gt, x1_gt:x2_gt] = 1
mask_regress[k,:,:] = torch.from_numpy(mask_regress_np)
mask_gt[k,:,:] = torch.from_numpy(mask_gt_np)

loss_regression = criterion2(mask_regress, mask_gt)
loss_regression.requires_grad = True
loss_labeling.backward(retain_graph=True)
loss_regression.backward()
optimizer.step()
...
```