Bounding Box Prediction from Scratch using PyTorch

Multi-Task learning — Bounding Box Regression + Image Classification

Aakanksha NS Jul 7, 2020 · 5 min read

Image clicked by author

Object detection is a very popular task in Computer Vision, where, given an image, you predict (usually rectangular) boxes around objects present in the image and also recognize the types of objects. There could be multiple objects in your image and there are various state-of-the-art techniques and architectures to tackle this problem like Faster-RCNN and YOLO v3.

This article talks about the case when there is only one object of interest present in an image. The focus here is more on how to read an image and its bounding box, resize and perform augmentations correctly, rather than on the model itself. The goal is to have a good grasp of the fundamental ideas behind object detection, which you can extend to get a better understanding of the more complex techniques.

Here's a link to the notebook consisting of all the code I've used for this article: https://jovian.ml/aakanksha-ns/road-signs-bounding-box-prediction

If you're new to Deep Learning or PyTorch, or just need a refresher, this might interest you:

Problem Statement

Given an image consisting of a road sign, predict a bounding box around the road sign and identify the type of road sign.

There are four distinct classes these signs could belong to:

- Traffic Light
- Stop
- Speed Limit
- Crosswalk

This is called a multi-task learning problem as it involves performing two tasks — 1) regression to find the bounding box coordinates, 2) classification to identify the type of road sign

Sample images. Source

Dataset

I've used the Road Sign Detection Dataset from Kaggle:

Road Sign Detection
877 images belonging to 4 classes.
www.kaggle.com

It consists of 877 images. It's a pretty imbalanced dataset, with most images belonging to the `speed limit` class, but since we're more focused on the bounding box prediction, we can ignore the imbalance.

Loading the Data

The annotations for each image were stored in separate `XML` files. I followed the following steps to create the training dataframe:

- Walk through the training directory to get a list of all the `.xml` files.
- Parse the `.xml` file using `xml.etree.ElementTree`
- Create a dictionary consisting of `filepath`, `width`, `height`, the bounding box coordinates (`xmin`, `xmax`, `ymin`, `ymax`) and `class` for each image and append the dictionary to a list.
- Create a pandas dataframe using the list of dictionaries of image stats.

```
def filelist(root, file_type):
    """Returns a fully-qualified list of filenames under root directory"""
    return [os.path.join(directory_path, f) for directory_path, directory_name
            files in os.walk(root) for f in files if f.endswith(file_type)]

def generate_train_df (anno_path):
    annotations = filelist(anno_path, '.xml')
    anno_list = []
    for anno_path in annotations:
        root = ET.parse(anno_path).getroot()
        anno = {}
        anno['filename'] = Path(str(images_path) + '/'+ root.find("./filename"
        anno['width'] = root.find("./size/width").text
        anno['height'] = root.find("./size/height").text
        anno['class'] = root.find("./object/name").text
        anno['xmin'] = int(root.find("./object/bndbox/xmin").text)
        anno['ymin'] = int(root.find("./object/bndbox/ymin").text)
        anno['xmax'] = int(root.find("./object/bndbox/xmax").text)
        anno['ymax'] = int(root.find("./object/bndbox/ymax").text)
        anno_list.append(anno)
    return pd.DataFrame(anno_list)
```

Hosted on Jovian View File

- Label encode the `class` column

```
#label encode target
class_dict = {'speedlimit': 0, 'stop': 1, 'crosswalk': 2, 'trafficlight': 3}
df_train['class'] = df_train['class'].apply(lambda x:  class_dict[x])
```

Hosted on Jovian View File

```
print(df_train.shape)
df_train.head()
```

```
(877, 8)
```

	filename	width	height	class	xmin	ymin	xmax	ymax
0	road_signs/images/road685.png	300	400	0	84	190	168	274
1	road_signs/images/road834.png	300	400	0	121	144	156	179
2	road_signs/images/road496.png	300	400	0	47	165	72	201
3	road_signs/images/road327.png	300	400	0	92	173	153	234
4	road_signs/images/road135.png	400	247	2	218	75	363	164

Hosted on Jovian View File

Resizing Images and Bounding Boxes

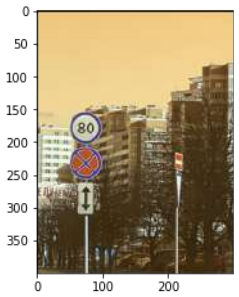
Since training a computer vision model needs images to be of the same size, we need to resize our images and their corresponding bounding boxes. Resizing an image is straightforward but resizing the bounding box is a little tricky because each box is relative to an image and its dimensions.

Here's how resizing a bounding box works:

- Convert the bounding box into an image (called mask) of the same size as the image it corresponds to. This mask would just have `0` for background and `1` for the area covered by the bounding box.

```
plt.imshow(im)
```

```
<matplotlib.image.AxesImage at 0x7ff2deb3bc88>
```

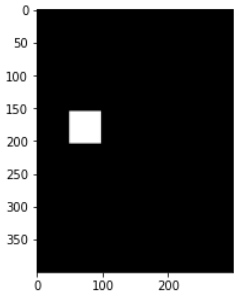


Hosted on Jovian View File

Original Image

```
plt.imshow(Y, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7ff2de689588>
```



Hosted on Jovian View File

Mask of the bounding box

- Resize the mask to the required dimensions.
- Extract bounding box coordinates from the resized mask.

```
def create_mask(bb, x):
    """Creates a mask for the bounding box of same shape as image"""
    rows,cols,*_ = x.shape
    Y = np.zeros((rows, cols))
    bb = bb.astype(np.int)
    Y[bb[0]:bb[2], bb[1]:bb[3]] = 1.
    return Y

def mask_to_bb(Y):
    """Convert mask Y to a bounding box, assumes 0 as background nonzero objec
    cols, rows = np.nonzero(Y)
    if len(cols)==0:
        return np.zeros(4, dtype=np.float32)
    top_row = np.min(rows)
    left_col = np.min(cols)
    bottom_row = np.max(rows)
    right_col = np.max(cols)
```

```
    return np.array([left_col, top_row, right_col, bottom_row], dtype=np.float

def create_bb_array(x):
    """Generates bounding box array from a train_df row"""
    return np.array([x[5],x[4],x[7],x[6]])
```

Hosted on Jovian View File

Helper functions to create mask from bounding box, extract bounding box coordinates from a mask

```
def resize_image_bb(read_path,write_path,bb,sz):
    """Resize an image and its bounding box and write image to new path"""
    im = read_image(read_path)
    im_resized = cv2.resize(im, (int(1.49*sz), sz))
    Y_resized = cv2.resize(create_mask(bb, im), (int(1.49*sz), sz))
    new_path = str(write_path/read_path.parts[-1])
    cv2.imwrite(new_path, cv2.cvtColor(im_resized, cv2.COLOR_RGB2BGR))
    return new_path, mask_to_bb(Y_resized)
```

Hosted on Jovian View File

Function to resize an image, write to a new path, and get resized bounding box coordinates

```
#Populating Training DF with new paths and bounding boxes
new_paths = []
new_bbs = []
train_path_resized = Path('./road_signs/images_resized')
for index, row in df_train.iterrows():
    new_path,new_bb = resize_image_bb(row['filename'], train_path_resized, cre
    new_paths.append(new_path)
    new_bbs.append(new_bb)
df_train['new_path'] = new_paths
df_train['new_bb'] = new_bbs
```

Hosted on Jovian View File

Data Augmentation

Data Augmentation is a technique to generalize our model better by creating new training images by using different variations of the existing images. We have only 800 images in our current training set, so data augmentation is very important to ensure our model doesn't overfit.

For this problem, I've used flip, rotation, center crop and random crop. I've talked about various data augmentation techniques in this article:

Image Processing Techniques for Computer Vision

Image Processing is an integral part of Computer vision. We almost always want to resize images, do data augmentation...

towardsdatascience.com

The only thing to remember here is ensuring that the bounding box is also transformed the same way as the image. To do this we follow the same approach as resizing — convert bounding box to a mask, apply the same transformations to the mask as the original image, and extract the bounding box coordinates.

```
# modified from fast.ai
def crop(im, r, c, target_r, target_c):
    return im[r:r+target_r, c:c+target_c]

# random crop to the original size
```

```
# random crop to the original size
def random_crop(x, r_pix=8):
    """ Returns a random crop"""
    r, c,*_ = x.shape
    c_pix = round(r_pix*c/r)
    rand_r = random.uniform(0, 1)
    rand_c = random.uniform(0, 1)
    start_r = np.floor(2*rand_r*r_pix).astype(int)
    start_c = np.floor(2*rand_c*c_pix).astype(int)
    return crop(x, start_r, start_c, r-2*r_pix, c-2*c_pix)

def center_crop(x, r_pix=8):
    r, c,*_ = x.shape
    c_pix = round(r_pix*c/r)
    return crop(x, r_pix, c_pix, r-2*r_pix, c-2*c_pix)
```

Hosted on Jovian View File

Helper functions to center crop and random crop an image

```
def rotate_cv(im, deg, y=False, mode=cv2.BORDER_REFLECT, interpolation=cv2.INT
    """ Rotates an image by deg degrees"""
    r,c,*_ = im.shape
    M = cv2.getRotationMatrix2D((c/2,r/2),deg,1)
    if y:
        return cv2.warpAffine(im, M,(c,r), borderMode=cv2.BORDER_CONSTANT)
    return cv2.warpAffine(im,M,(c,r), borderMode=mode, flags=cv2.WARP_FILL_OUT

def random_cropXY(x, Y, r_pix=8):
    """ Returns a random crop"""
    r, c,*_ = x.shape
    c_pix = round(r_pix*c/r)
    rand_r = random.uniform(0, 1)
    rand_c = random.uniform(0, 1)
    start_r = np.floor(2*rand_r*r_pix).astype(int)
    start_c = np.floor(2*rand_c*c_pix).astype(int)
    xx = crop(x, start_r, start_c, r-2*r_pix, c-2*c_pix)
    YY = crop(Y, start_r, start_c, r-2*r_pix, c-2*c_pix)
    return xx, YY

def transformsXY(path, bb, transforms):
    x = cv2.imread(str(path)).astype(np.float32)
    x = cv2.cvtColor(x, cv2.COLOR_BGR2RGB)/255
    Y = create_mask(bb, x)
    if transforms:
        rdeg = (np.random.random()-.50)*20
        x = rotate_cv(x, rdeg)
        Y = rotate_cv(Y, rdeg, y=True)
        if np.random.random() > 0.5:
            x = np.fliplr(x).copy()
            Y = np.fliplr(Y).copy()
        x, Y = random_cropXY(x, Y)
    else:
        x, Y = center_crop(x), center_crop(Y)
    return x, mask_to_bb(Y)
```

Hosted on Jovian View File

Transforming image and mask

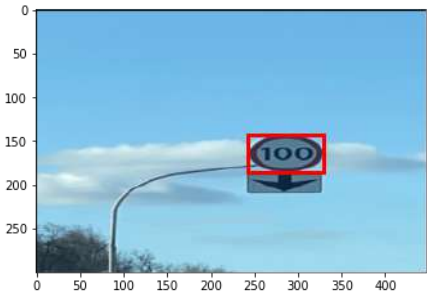
```
def create_corner_rect(bb, color='red'):
    bb = np.array(bb, dtype=np.float32)
    return plt.Rectangle((bb[1], bb[0]), bb[3]-bb[1], bb[2]-bb[0], color=color
                         fill=False, lw=3)

def show_corner_bb(im, bb):
    plt.imshow(im)
    plt.gca().add_patch(create_corner_rect(bb))
```

Hosted on Jovian View File

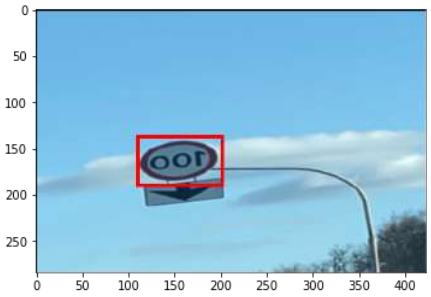
Displaying bounding box

```
#original
im = cv2.imread(str(df_train.values[68][8]))
im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
show_corner_bb(im, df_train.values[68][9])
```



Hosted on Jovian View File

```
# after transformation
im, bb = transformsXY(str(df_train.values[68][8]),df_train.values[68][9],True
show_corner_bb(im, bb)
```



Hosted on Jovian View File

PyTorch Dataset

Now that we have our data augmentations in place, we can do the train-validation split and create our PyTorch dataset. We normalize the images using ImageNet stats because we're using a pre-trained ResNet model and apply data augmentations in our dataset while training.

```
X_train, X_val, y_train, y_val = train_test_split(X, Y, test_size=0.2, random_
```

Hosted on Jovian View File

train-valid split

```
def normalize(im):
    """Normalizes images with Imagenet stats."""
    imagenet_stats = np.array([[0.485, 0.456, 0.406], [0.229, 0.224, 0.225]])
    return (im - imagenet_stats[0])/imagenet_stats[1]
```

Hosted on Jovian View File

```
class RoadDataset(Dataset):
    def __init__(self, paths, bb, y, transforms=False):
        self.transforms = transforms
        self.paths = paths.values
        self.bb = bb.values
        self.y = y.values
    def __len__(self):
        return len(self.paths)

    def __getitem__(self, idx):
        path = self.paths[idx]
        y_class = self.y[idx]
        x, y_bb = transformsXY(path, self.bb[idx], self.transforms)
```

```
        x = normalize(x)
        x = np.rollaxis(x, 2)
        return x, y_class, y_bb
```

Hosted on Jovian View File

```
train_ds = RoadDataset(X_train['new_path'],X_train['new_bb'] ,y_train, transfo
valid_ds = RoadDataset(X_val['new_path'],X_val['new_bb'],y_val)
```

Hosted on Jovian View File

Creating train and valid datasets

```
batch_size = 64
train_dl = DataLoader(train_ds, batch_size=batch_size, shuffle=True)
valid_dl = DataLoader(valid_ds, batch_size=batch_size)
```

Hosted on Jovian View File

Setting the batch size and creating data loaders

PyTorch Model

For the model, I've used a very simple pre-trained resNet-34 model. Since we have two tasks to accomplish here, there are two final layers — the bounding box regressor and the image classifier.

```
class BB_model(nn.Module):
    def __init__(self):
        super(BB_model, self).__init__()
        resnet = models.resnet34(pretrained=True)
        layers = list(resnet.children())[:8]
        self.features1 = nn.Sequential(*layers[:6])
        self.features2 = nn.Sequential(*layers[6:])
        self.classifier = nn.Sequential(nn.BatchNorm1d(512), nn.Linear(512, 4)
        self.bb = nn.Sequential(nn.BatchNorm1d(512), nn.Linear(512, 4))

    def forward(self, x):
        x = self.features1(x)
        x = self.features2(x)
        x = F.relu(x)
        x = nn.AdaptiveAvgPool2d((1,1))(x)
        x = x.view(x.shape[0], -1)
        return self.classifier(x), self.bb(x)
```

Hosted on Jovian View File

Training

For the loss, we need to take into both classification loss and the bounding box regression loss, so we use a combination of cross-entropy and L1-loss (sum of all the absolute differences between the true value and the predicted coordinates). I've scaled the L1-loss by a factor of 1000 because to have both the classification and regression losses in a similar range. Apart from this, it's a standard PyTorch training loop (using a GPU):

```
def update_optimizer(optimizer, lr):
    for i, param_group in enumerate(optimizer.param_groups):
        param_group["lr"] = lr
```

Hosted on Jovian View File

```
def train_epocs(model, optimizer, train_dl, val_dl, epochs=10,C=1000):
    idx = 0
    for i in range(epochs):
        model.train()
        total = 0
        sum_loss = 0
        for x, y_class, y_bb in train_dl:
            batch = y_class.shape[0]
            x = x.cuda().float()
            y_class = y_class.cuda()
            y_bb = y_bb.cuda().float()
            out_class, out_bb = model(x)
            loss_class = F.cross_entropy(out_class, y_class, reduction="sum")
            loss_bb = F.l1_loss(out_bb, y_bb, reduction="none").sum(1)
            loss_bb = loss_bb.sum()
            loss = loss_class + loss_bb/C
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            idx += 1
            total += batch
            sum_loss += loss.item()
        train_loss = sum_loss/total
        val_loss, val_acc = val_metrics(model, valid_dl, C)
        print("train_loss %.3f val_loss %.3f val_acc %.3f" % (train_loss, val_
    return sum_loss/total
```

Hosted on Jovian View File

```
def val_metrics(model, valid_dl, C=1000):
    model.eval()
    total = 0
    sum_loss = 0
    correct = 0
    for x, y_class, y_bb in valid_dl:
        batch = y_class.shape[0]
        x = x.cuda().float()
        y_class = y_class.cuda()
        y_bb = y_bb.cuda().float()
        out_class, out_bb = model(x)
        loss_class = F.cross_entropy(out_class, y_class, reduction="sum")
        loss_bb = F.l1_loss(out_bb, y_bb, reduction="none").sum(1)
        loss_bb = loss_bb.sum()
        loss = loss_class + loss_bb/C
        _, pred = torch.max(out_class, 1)
        correct += pred.eq(y_class).sum().item()
        sum_loss += loss.item()
        total += batch
    return sum_loss/total, correct/total
```

Hosted on Jovian View File

```
model = BB_model().cuda()
parameters = filter(lambda p: p.requires_grad, model.parameters())
optimizer = torch.optim.Adam(parameters, lr=0.006)
```

Hosted on Jovian View File

```
train_epocs(model, optimizer, train_dl, valid_dl, epochs=15)
```

```
train_loss 2.476 val_loss 51187.074 val_acc 0.119
train_loss 1.661 val_loss 998.014 val_acc 0.091
train_loss 1.657 val_loss 54.162 val_acc 0.261
train_loss 1.478 val_loss 40.076 val_acc 0.432
train_loss 1.228 val_loss 1.634 val_acc 0.688
train_loss 1.066 val_loss 1.338 val_acc 0.631
train_loss 0.938 val_loss 0.970 val_acc 0.761
train_loss 0.852 val_loss 1.103 val_acc 0.699
train_loss 0.867 val_loss 0.845 val_acc 0.773
train_loss 0.811 val_loss 0.913 val_acc 0.739
train_loss 0.803 val_loss 0.812 val_acc 0.761
train_loss 0.766 val_loss 0.847 val_acc 0.773
train_loss 0.822 val_loss 0.941 val_acc 0.739
```

```
train_loss 0.801 val_loss 2.106 val_acc 0.767
```

```
0.8008461760386251
```

Hosted on Jovian View File

Prediction on Test Images

Now that we're done with training, we can pick a random image and test our model on it. Even though we had a fairly small number of training images, we end up getting a pretty decent prediction on our test image.

It'll be a fun exercise to take a real photo using your phone and test out the model. Another interesting experiment would be to not perform any data augmentations and train the model and compare the two models.

```
# resizing test image
im = read_image('./road_signs/images_resized/road789.png')
im = cv2.resize(im, (int(1.49*300), 300))
cv2.imwrite('./road_signs/road_signs_test/road789.jpg', cv2.cvtColor(im, cv2.C
```

```
True
```

Hosted on Jovian View File

```
# test Dataset
test_ds = RoadDataset(pd.DataFrame([{'path':'./road_signs/road_signs_test/roac
x, y_class, y_bb = test_ds[0]
```

Hosted on Jovian View File

```
xx = torch.FloatTensor(x[None,])
xx.shape
```

```
torch.Size([1, 3, 284, 423])
```

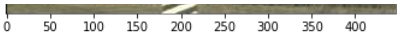
Hosted on Jovian View File

```
# prediction
out_class, out_bb = model(xx.cuda())
out_class, out_bb
```

```
(tensor([[ 4.0389, -0.7852, -1.1573, -0.6184]], device='cuda:0',
        grad_fn=<AddmmBackward>),
 tensor([[120.0455, 176.9376, 157.2672, 247.1257]], device='cuda:0',
        grad_fn=<AddmmBackward>))
```

Hosted on Jovian View File

```
# predicted bounding box
bb_hat = out_bb.detach().cpu().numpy()
bb_hat = bb_hat.astype(int)
show_corner_bb(im, bb_hat[0])
```



Hosted on Jovian View File

Conclusion

Now that we've covered the fundamentals of object detection and implemented it from scratch, you can extend these ideas to the multi-object case and try out more complex models like RCNN and YOLO! Also, check out this super cool library called albumentations to perform data augmentations easily.

References

- Deep Learning summer elective at the University of San Francisco's Master's in Data Science program
- https://www.usfca.edu/data-institute/certificates/fundamentals-deep-learning