Object Detection and Localization with Deep Networks

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Wednesday 3rd June, 2020 10:59
Object detection in images is a more difficult problem than the problem of 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.

In this lecture, we will assume that an image has only one object in it. The job of the CNN is to recognize the category of the object and to estimate the coordinates of the smallest bounding-box rectangle that contains the object.

Estimating the bounding-box rectangle is referred to as regression.

So our goal is to design a convolutional network that can make two inferences simultaneously, one for classification and the other for regression.
It follows that our convolutional network must use two loss functions, one for classification and the other for regression.

Backpropagating two losses through a network raises interesting issues related to the programming involved and also whether the gradients of the two losses with respect to the learnable parameters that are in common between the two inference paths can somehow “interfere” with one another.

Regarding the programming issue raised by using two loss functions, as you know, ordinarily when one calls `backwards()` on a loss, that causes the computational graph constructed during the forward propagation to be dismantled. But we obviously cannot allow for that to happen when using two loss functions.

The goal of this lecture is to present a convolutional network that carries out both the classification and the regression simultaneously.
Obviously, training such networks requires image data that must include bounding-box annotations in addition to the object labels.

This lecture will also introduce you to a new dataset, PurdueShapes5, of 32x32 images that I have created for experimenting with object detection and localization problems. Associated with each image is the label of the object in the image and also the coordinates of the bounding box rectangle for the object.

As you would expect, any new dataset for training a CNN calls for a custom dataloader. A dataloader for the PurdueShapes5 dataset is included in Version 1.0.7 of the DLStudio module.
A Dual-Inferencing Convolutional Network for Simultaneous Classification and Regression

PurdueShapes5 — A Dataset of Small-Sized Images to Experiment with Object Detection

A Custom Dataloader for PurdueShapes5

Creating a Network for Detecting and Localizing Objects

Training and Testing the LOADnet2 Network
Outline

1. A Dual-Inferencing Convolutional Network for Simultaneous Classification and Regression
2. PurdueShapes5 — A Dataset of Small-Sized Images to Experiment with Object Detection
3. A Custom Dataloader for PurdueShapes5
4. Creating a Network for Detecting and Localizing Objects
5. Training and Testing the LOADnet2 Network
Obviously, what we need to implement is a dual-inferencing CNN that has two different outputs for the same input image: one for classification and the other for regression.

The classification output must map the input image to its category label, and the regression output must map the same input to the coordinates of the bounding box rectangle.
Now We Need Two Different Loss Functions

For all of the classification work we have done so far, we have used for the loss the cross-entropy measure through the PyTorch class torch.nn.CrossEntropyLoss.

In what follows, I’ll argue that whereas the cross-entropy is great as a measure of the misclassification error, it doesn’t have the right properties for what is needed for the regression error.

Consider the classification of the CIFAR-10 images. The output layer for a CNN for this dataset will have 10 nodes, one for each of the 10 classes.

Let the vector \( \mathbf{x} \) represent the output for a given input image whose category label is \( c \), which is an integer between 0 and 9. The cross-entropy loss for this output would be given by

\[
cross_{-}\text{entropy}(\mathbf{x}, c) = - \log \frac{e^{x[c]}}{\sum_{j=0}^{9} e^{x[j]}}
\]
Appropriateness of Cross-Entropy Loss for Measuring Classification Error

- To see why the formula shown on the previous slide makes total sense for measuring the quality of the predicted label for a given input, first focus on the fact that, if the inferencing were to be perfect, only the output element \( x[c] \) would light up and all the other elements in the vector \( x \) would be 0. In this case, the loss as measured by the formula would be zero. We can therefore say that the error in the label prediction for the input image under consideration is proportional to the extent \( x \) satisfies this property.

- The question now is: What’s the best way to measure the above mentioned property for the output vector?

- To answer the question, let’s switch to a probabilistic interpretation of the output. The 10 output values given by the ratios \( \frac{e^x[c]}{\sum_{j=0}^{9} e^x[j]} \) for the 10 different value of \( c \) can be interpreted as the probabilities because the numerator is guaranteed to be positive regardless of the sign of the values \( x[c] \) and because these 10 numbers add up to 1.0.
The probabilistic interpretation of the output allows for it to be characterized by cross-entropy vis-a-vis the input.

In general, if $p([j])$ is a probabilistic characterization of the class label $j$ for the input image and $q(x[j])$ a probabilistic characterization of output as explained above, the cross-entropy between the two probability distributions would be given by

$$H(p, q) = - \sum_j p([j]) \cdot \log_2 q(x[j])$$

Now consider the case when we are sure that the class label for the input image is $c$, meaning that $p[j] = 1$ for $j = c$ and 0 otherwise, the above formula becomes

$$H(p, q) = - \log_2 q(x[c])$$
Cross-Entropy Loss for Measuring the Classification Error (contd.)

- To gain an even deeper understanding of the power of the cross-entropy criterion for measuring the classification error, it’s best to work out an example of $q(x[c])$ by hand.

- Let’s start by choosing a value for $c$. Let’s say $c = 4$ for the correct label for the input image.

- Now assign different values to the 10 elements of $q(x[j])$ and see what you get for the cross-entropy loss. For example, you could choose $q = (0.0, 0.2, 0.1, 0.3, 0.2, 0.2, 0.0, 0.0, 0.0)$. Keep in mind that the quantity $x \cdot \log x \rightarrow 0$ as $x \rightarrow 0$. Additionally, $\log x \rightarrow -\infty$ as $x \rightarrow 0$.

- You will notice that should the CNN produce a value of 0 for $x[4]$, the loss as calculated by cross-entropy would be very large indeed (theoretically approaching $\infty$), which would then propagate backwards through the network as the weights undergo significant changes in response to such a large loss.
Typically, if the cross-entropy loss exceeds 2.0, the predicted labels are mostly noise.

In PyTorch, the cross-entropy loss function shown at the bottom of Slide 8 is used through the class `torch.cc.CrossEntropyLoss`. Its documentation is provided at

https://pytorch.org/docs/stable/nn.html#crossentropyloss

While I am on the topic of using cross-entropies for measuring the label prediction loss, let’s also consider the special case when our classification involves only two classes. In this case, the cross-entropy formula shown on Slide 10 can be expressed as

\[
H(p, q) = - [p([0]) \cdot \log_2 q(x[0]) + p([1]) \cdot \log_2 q(x[1])] \\
= - [p \cdot \log_2 q + (1 - p) \cdot (1 - q)]
\]
In the equations shown at the bottom of the previous slide, the labels for the two classes are denoted ‘0’ and ‘1’. In the second equation, we have denoted $p[0]$ by $p$, which would make $p[1] = 1 - p$, and $q[0]$ by $q$, which would make $q[1] = 1 - q$.

As with the earlier multi-class formula, if you are certain that the input pattern belongs to class 0, the loss function shown above reduces to just $-\log_2 q$. On the other hand, if the input image definitely belongs to class 1, the loss becomes $-\log_2(1 - q)$.

In PyTorch, the binary version of the cross-entropy loss can be used through the class `torch.nn.BCELoss` where “BCELoss” stands for “Binary Cross Entropy Loss”. Here is the documentation page for this loss function:

https://pytorch.org/docs/stable/nn.html#bceloss
**But What About the Regression Loss?**

- While the cross-entropy loss takes care of the classification error, it’s not appropriate as a measure of the bounding-box regression error.

- Bounding-box regression is about the numerics of where exactly the object is in an image and requires a measure that is more geometrical in nature.

- **Bounding-box regression loss is best measured by the two loss functions illustrated on the next slide.**

- The acronym “IoU” stands for “Intersection over Union”. In problems that require measuring the similarity between two sets, this loss is more commonly known as the “Jaccard Distance”.

- The acronym “MSE” stands for the “Mean-Squared Error”.
You can use the `torch.nn.MSELoss` class for measuring the MSE loss.

However, you have to program up the IoU loss yourself.
Outline

1. A Dual-Inferencing Convolutional Network for Simultaneous Classification and Regression

2. PurdueShapes5 — A Dataset of Small-Sized Images to Experiment with Object Detection

3. A Custom Dataloader for PurdueShapes5

4. Creating a Network for Detecting and Localizing Objects

5. Training and Testing the LOADnet2 Network
The PurdueShapes5 Dataset with Bbox Annotations

- Before I demonstrate how to write code for implementing in PyTorch the network architecture shown in Slide 7, let me first talk about a dataset I have created for CNN training that involves both the class labels and the bounding box 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 $32 \times 32$) (or, perhaps, $64 \times 64$ in the future) for demonstrating concepts related to object detection and localization.

- So I have created the PurdueShapes5 dataset to fill this void.

- The program that generates the dataset also generates the bounding-box (bbox) annotations for the objects.
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 “Examples” directory 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 $32 \times 32$ image in the dataset is stored using 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

- Each shape generated for the dataset is subject to randomization with respect to its size, its orientation, and its exact location in the image frame. Since the orientation randomization is carried out with a very simple non-interpolating transform, just the act of random rotations can introduce boundary and even interior noise in the patterns.

- I serialize the dataset with Python’s pickle module and then compress it with Python’s gzip module.
Extracting the Pixels and the Bbox from the Images

- 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:

  ```python
data = list(im.getdata())  # 'im' is an object of type Image
R = [pixel[0] for pixel in data]  # data for the input channels
G = [pixel[1] for pixel in data]
```

  ```python
## 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:

  ```python
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. A Dual-Inferencing Convolutional Network for Simultaneous Classification and Regression
2. PurdueShapes5 — A Dataset of Small-Sized Images to Experiment with Object Detection
3. A Custom Dataloader for PurdueShapes5
4. Creating a Network for Detecting and Localizing Objects
5. Training and Testing the LOADnet2 Network
Creating a custom dataloader 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.

The new inner class CustomDataLoading of the DLStudio module presents a custom dataloader for the PurdueShapes5 dataset. This dataloader understands the data format presented on Slide 21.

The next slide presents the implementation of the dataloader. Note that in the last two statements on the next slide, the arguments dataserver_train and dataserver_test are both instances of the class PurdueShapes5Dataset. One of these points to where the training data is and the other that points to where the test data is.
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):
        return len(self.dataset)

    def __getitem__(self, idx):
        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 = tvt.Compose([tvt.ToTensor(),
                             tvt.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=4)
    self.test_data_loader = torch.utils.data.DataLoader(dataset_server_test,
                                                        batch_size=self.dl_studio.batch_size, shuffle=False, num_workers=4)
Outline

1. A Dual-Inferencing Convolutional Network for Simultaneous Classification and Regression
2. PurdueShapes5 — A Dataset of Small-Sized Images to Experiment with Object Detection
3. A Custom Dataloader for PurdueShapes5
4. Creating a Network for Detecting and Localizing Objects
5. Training and Testing the LOADnet2 Network
The LOADnet (LOcalizing And Detecting Network) 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 want to predict the exact locations of the corners, perhaps being at the same abstraction as for the labeling part is not even desirable.

- The next two slides present the LOADnet2 network that I have worked with the most for developing Version 1.0.7 of DLStudio.
The LOADnet2 Network

class LOADnet2(nn.Module):
    """
The acronym 'LOAD' stands for 'LOcalization And Detection'.
LOADnet2 uses both convo and linear layers for regression
"""
def __init__(self, skip_connections=True, depth=8):
    super(DLStudio.DetectAndLocalize.LOADnet2, self).__init__()
    if depth not in [8,10,12,14,16]:
        sys.exit("LOADnet2 has only been tested for 'depth' values 8, 10, 12, 14, and 16")
    self.depth = depth // 2
    self.conv = nn.Conv2d(3, 64, 3, padding=1)
    self.pool = nn.MaxPool2d(2, 2)
    self.bn1 = nn.BatchNorm2d(64)
    self.bn2 = nn.BatchNorm2d(128)
    self.skip64_arr = nn.ModuleList()
    for i in range(self.depth):
        self.skip64_arr.append(DLStudio.DetectAndLocalize.SkipBlock(64, 64, skip_connections=skip_connections))
    self.skip64ds = DLStudio.DetectAndLocalize.SkipBlock(64, 64, downsample=True, skip_connections=skip_connections)
    self.skip64to128 = DLStudio.DetectAndLocalize.SkipBlock(64, 128, skip_connections=skip_connections)
    self.skip128_arr = nn.ModuleList()
    for i in range(self.depth):
        self.skip128_arr.append(DLStudio.DetectAndLocalize.SkipBlock(128, 128, skip_connections=skip_connections))
    self.skip128ds = DLStudio.DetectAndLocalize.SkipBlock(128, 128, downsample=True, skip_connections=skip_connections)
    self.fc1 = nn.Linear(2048, 1000)
    self.fc2 = nn.Linear(1000, 10)
    ## for regression
    self.conv_seqn = nn.Sequential(
        nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU(inplace=True),
        nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
        nn.ReLU(inplace=True))

(Continued on the next slide .....)

Purdue University 28
Creating a Network for Detecting and Localizing Objects

The LOADnet2 Network (contd.)

(... continued from the previous slide)

```python
self.fc_seqn = nn.Sequential(
    nn.Linear(16384, 1024),
    nn.ReLU(inplace=True),
    nn.Linear(1024, 512),
    nn.ReLU(inplace=True),
    nn.Linear(512, 4)
)

def forward(self, x):
    x = self.pool(torch.nn.functional.relu(self.conv(x)))
    ## THE LABELING SECTION:
    x1 = x.clone()
    for i,skip64 in enumerate(self.skip64_arr[:self.depth//4]):
        x1 = skip64(x1)
    x1 = self.skip64ds(x1)
    for i,skip64 in enumerate(self.skip64_arr[self.depth//4:]):
        x1 = skip64(x1)
    x1 = self.bn1(x1)
    x1 = self.skip64to128(x1)
    for i,skip128 in enumerate(self.skip128_arr[:self.depth//4]):
        x1 = skip128(x1)
    x1 = self.bn2(x1)
    x1 = self.skip128ds(x1)
    for i,skip128 in enumerate(self.skip128_arr[self.depth//4:]):
        x1 = skip128(x1)
    x1 = x1.view(-1, 2048)
    x1 = torch.nn.functional.relu(self.fc1(x1))
    x1 = self.fc2(x1)

    ## THE REGRESSION SECTION:
    x2 = self.conv_seqn(x)
    x2 = self.conv_seqn(x2)
    # flatten
    x2 = x2.view(x.size(0), -1)
    x2 = self.fc_seqn(x2)
    return x1, x2
```

Purdue University
Outline

1. A Dual-Inferencing Convolutional Network for Simultaneous Classification and Regression
2. PurdueShapes5 — A Dataset of Small-Sized Images to Experiment with Object Detection
3. A Custom Dataloader for PurdueShapes5
4. Creating a Network for Detecting and Localizing Objects
5. Training and Testing the LOADnet2 Network
Training the LOADnet2 Network

def run_code_for_training_with_CrossEntropy_and_MSE_Losses(self, net):
    ...
    net = net.to(self_dl_studio.device)
criterion1 = nn.CrossEntropyLoss()
criterion2 = nn.MSELoss()
    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):
            inputs, bbox_gt, labels = data['image'], data['bbox'], data['label']
            if self_dl_studio.debug_train and i % 500 == 499:
                print("\n\n\n[epoch=%d iter=%d:]
% Ground Truth: 
\n% (epoch+1, i+1) +
\n% .join(\%10s % self.dataserver_train.class_labels[labels[j].item()] for j in range(self_dl_studio.batch_size)))
            inputs = inputs.to(self_dl_studio.device)
            labels = labels.to(self_dl_studio.device)
            bbox_gt = bbox_gt.to(self_dl_studio.device)
            optimizer.zero_grad()
            outputs = net(inputs)
            outputs_label = outputs[0]  # prediction from the classification side
            bbox_pred = outputs[1]  # prediction from the regression side

            loss_labeling = criterion1(outputs_label, labels)
            loss_labeling.backward(retain_graph=True)
            loss_regression = criterion2(bbox_pred, bbox_gt)
            loss_regression.backward()
            optimizer.step()
            running_loss_labeling += loss_labeling.item()
            running_loss_regression += loss_regression.item()
            ## code for displaying intermediate results

Purdue University
Training and Testing the LOADnet2 Network

The Two Losses vs. the Iterations During Training

[epoch=1 iter=1:] Ground Truth: star disk oval rectangle
predicted Labels: oval disk disk disk
gt_bb: [2,2,17,18] pred_bb: [0,0,0,0]
gt_bb: [12,2,22,12] pred_bb: [0,0,0,0]
gt_bb: [8,0,24,19] pred_bb: [0,0,0,0]
gt_bb: [3,0,5,3] pred_bb: [0,0,0,0]

[epoch=1 iter=10:] Ground Truth: star oval oval star
predicted Labels: rectangle rectangle rectangle rectangle
gt_bb: [5,3,22,20] pred_bb: [0,0,0,0]
gt_bb: [14,0,25,16] pred_bb: [0,0,0,0]
gt_bb: [15,5,27,18] pred_bb: [0,0,0,0]
gt_bb: [3,9,16,22] pred_bb: [0,0,0,0]
Training and Testing the LOADnet2 Network

The Two Losses vs. the Iterations During Training (contd.)

[epoch=1 iter=100:] Ground Truth: triangle rectangle oval oval
[epoch=1 iter=100:] Predicted Labels: triangle star star disk

gt_bb: [2,0,23,28]
pred_bb: [0,0,0,0]
gt_bb: [3,2,8,7]
pred_bb: [0,0,0,0]
gt_bb: [9,0,23,18]
pred_bb: [0,0,0,0]
gt_bb: [21,18,31,28]
pred_bb: [0,0,0,0]

[epoch=1 iter=500:] Ground Truth: star rectangle triangle disk
[epoch=1 iter=500:] Predicted Labels: triangle rectangle triangle disk

gt_bb: [3,12,20,29]
pred_bb: [8,5,22,19]
gt_bb: [3,0,22,18]
pred_bb: [7,5,19,16]
gt_bb: [4,1,25,28]
pred_bb: [10,7,28,23]
gt_bb: [4,6,20,22]
pred_bb: [8,5,21,18]

Training and Testing the LOADnet2 Network

The Two Losses vs. the Iterations During Training (contd.)

[epoch=1 iter=1000:] Ground Truth: oval oval oval rectangle
[epoch=1 iter=1000:] Predicted Labels: oval oval oval oval
  gt_bb: [9,7,20,10]
  pred_bb: [4,3,14,13]
  gt_bb: [14,5,31,20]
  pred_bb: [8,6,27,25]
  gt_bb: [13,3,23,21]
  pred_bb: [7,5,24,21]
  gt_bb: [16,9,30,21]
  pred_bb: [9,6,29,26]

[epoch:1, iteration: 1000] loss_labeling: 0.636 loss_regression: 20.696

[epoch=1 iter=1500:] Ground Truth: oval oval disk star
[epoch=1 iter=1500:] Predicted Labels: oval oval disk star
  gt_bb: [15,7,30,17]
  pred_bb: [14,6,26,17]
  gt_bb: [19,10,31,24]
  pred_bb: [18,12,28,22]
  gt_bb: [12,2,26,16]
  pred_bb: [10,1,25,16]
  gt_bb: [0,5,9,15]
  pred_bb: [1,5,9,14]

[epoch:1, iteration: 1500] loss_labeling: 0.434 loss_regression: 8.058
Training and Testing the LOADnet2 Network

The Two Losses vs. the Iterations During Training (contd.)

[epoch=1 iter=2000:] Ground Truth: rectangle star triangle disk
Predicted Labels: oval star triangle disk

gt_bb: [11,4,31,25]
pred_bb: [9,5,28,23]

gt_bb: [7,0,20,9]
pred_bb: [7,0,16,9]

gt_bb: [3,6,28,29]
pred_bb: [3,5,25,27]

gt_bb: [8,4,22,18]
pred_bb: [7,3,20,17]

[epoch=1, iteration: 2000] loss_labeling: 0.345 loss_regression: 2.526

[epoch=1 iter=2500:] Ground Truth: triangle triangle triangle disk
Predicted Labels: triangle triangle triangle disk

gt_bb: [1,4,27,31]
pred_bb: [1,3,26,30]

gt_bb: [1,3,28,29]
pred_bb: [1,4,25,29]

gt_bb: [0,6,20,31]
pred_bb: [0,7,19,28]

gt_bb: [7,18,21,31]
pred_bb: [7,18,21,31]

[epoch=1, iteration: 2500] loss_labeling: 0.324 loss_regression: 2.210
Training and Testing the LOADnet2 Network

The Two Losses vs. the Iterations During Training (contd.)

[epoch=2 iter=500:] Ground Truth: triangle rectangle triangle oval
[epoch=2 iter=500:] Predicted Labels: triangle rectangle triangle oval

gt_bb: [0,1,24,26]
pred_bb: [0,0,24,25]

gt_bb: [5,0,24,17]
pred_bb: [6,0,24,17]

gt_bb: [0,5,28,31]
pred_bb: [2,3,29,31]

gt_bb: [12,0,19,14]
pred_bb: [11,1,21,12]

[epoch:2, iteration: 500] loss_labeling: 0.242 loss_regression: 1.722

[epoch=2 iter=1000:] Ground Truth: star rectangle oval rectangle
[epoch=2 iter=1000:] Predicted Labels: star rectangle oval rectangle

gt_bb: [3,4,14,14]
pred_bb: [3,2,13,13]

gt_bb: [12,10,31,22]
pred_bb: [14,9,30,22]

gt_bb: [18,13,29,21]
pred_bb: [18,12,29,21]

gt_bb: [16,16,25,24]
pred_bb: [16,15,25,24]

[epoch:2, iteration: 1000] loss_labeling: 0.231 loss_regression: 1.437
Training and Testing the LOADnet2 Network

The Two Losses vs. the Iterations During Training (contd.)

[epoch=2 iter=1500:] Ground Truth: oval disk triangle triangle
[epoch=2 iter=1500:] Predicted Labels: disk disk triangle triangle

gt_bb: [15,13,31,21]
pred_bb: [16,12,28,22]

[epoch=2 iter=2500:] Ground Truth: triangle disk rectangle rectangle
[epoch=2 iter=2500:] Predicted Labels: triangle disk rectangle oval

gt_bb: [7,6,30,28]
pred_bb: [6,5,30,29]
Results on Unseen Test Data

[i=0:] Ground Truth: rectangle triangle disk oval
Predicted Labels: rectangle triangle disk oval

\[
\begin{align*}
\text{gt_bb: } & [19,15,27,22] \\
\text{pred_bb: } & [19,13,26,21] \\
\text{gt_bb: } & [2,0,31,28] \\
\text{pred_bb: } & [2,0,30,28] \\
\text{gt_bb: } & [4,21,14,31] \\
\text{pred_bb: } & [4,19,14,30] \\
\text{gt_bb: } & [7,6,19,10] \\
\text{pred_bb: } & [8,3,17,10]
\end{align*}
\]
Classification Accuracy on the Unseen Test Data (After 2 Epochs of Training)

Prediction accuracy for rectangle : 80 %
Prediction accuracy for triangle : 99 %
Prediction accuracy for disk : 100 %
Prediction accuracy for oval : 77 %
Prediction accuracy for star : 99 %

Overall accuracy of the network on the 1000 test images: 91 %

Displaying the confusion matrix:

<table>
<thead>
<tr>
<th></th>
<th>rectangle</th>
<th>triangle</th>
<th>disk</th>
<th>oval</th>
<th>star</th>
</tr>
</thead>
<tbody>
<tr>
<td>rectangle:</td>
<td>80.00</td>
<td>0.00</td>
<td>0.00</td>
<td>20.00</td>
<td>0.00</td>
</tr>
<tr>
<td>triangle:</td>
<td>0.00</td>
<td>99.50</td>
<td>0.00</td>
<td>0.00</td>
<td>0.50</td>
</tr>
<tr>
<td>disk:</td>
<td>0.00</td>
<td>0.00</td>
<td>100.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>oval:</td>
<td>22.00</td>
<td>0.00</td>
<td>0.50</td>
<td>77.50</td>
<td>0.00</td>
</tr>
<tr>
<td>star:</td>
<td>0.00</td>
<td>0.50</td>
<td>0.00</td>
<td>0.00</td>
<td>99.50</td>
</tr>
</tbody>
</table>