Object Detection and Localization with Deep Networks

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Wednesday 24$^{\text{th}}$ March, 2021 20:53

Preamble

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.

Preamble (contd.)

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.

Preamble (contd.)

Obviously, training such networks requires image data that must include bounding-box annotations in addition to the object labels.

This lecture will also introduces 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 the DLStudio module.

Outline

Outline

A Dual-Inferencing CNN

- 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.

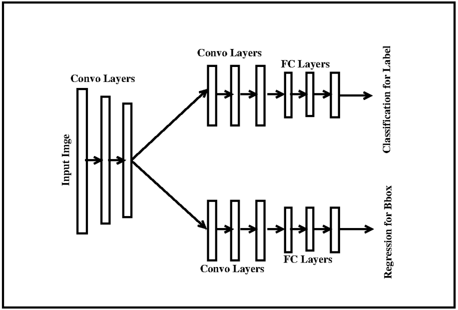


Figure: A dual-inferencing CNN

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 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

$$cross\_entropy(\mathrm{x}, c) \quad = \quad -\log \frac{e^{\mathrm{x}[c]}}{\sum_{j=0}^{9} e^{\mathrm{x}[j]}} \tag{1}$$

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.

- For perfect inference, the loss as measured by the formula would be zero. As to why, assume that by "lighting up" we mean that the value at the node indexed c is 1, while it is 0 at all the other nodes. In this case both the numerator and the denominator of the ratio in the formula would equal e, making the ratio equal to 1, and the log of 1 is always zero. Hence zero loss.

- Since the value of the loss would be 0 for the case of perfect inference as mentioned above, we are allowed to say that the error in predicting a label for the input image under consideration should be *proportional to the extent to which x departs from this property* for a real image.

Cross-Entropy Loss for Measuring Classification Error (contd.)

- The question now is: **What's the best way to measure the property of the output vector x of a neural network whose element values differ from what they would be for perfect inference as described at the bottom of the previous slide?**

- To answer the question, **let's switch to a probabilistic interpretation of the output values $x[i]$ for $i = 0, \ldots, 9$.** The 10 output values given by the ratios $\frac{e^{x[i]}}{\sum_{j=0}^{9} e^{x[j]}}$ for the 10 different value of the index i can be interpreted as probabilities because the numerator is guaranteed to be positive regardless of the sign of the values $x[i]$ and because these 10 numbers add up to 1.0.

- The probabilistic interpretation of the ratios $\frac{e^{x[i]}}{\sum_{j=0}^{9} e^{x[j]}}$ for $i = 0, \ldots, 9$ allows for them to be characterized by **cross-entropy** vis-a-vis the input, as explained over the next few slides.

Outline

Understanding Entropy

- Ideally, the starting point for understanding the notion of cross-entropy is the idea of entropy itself: Entropy is a measure of uncertainty and its value for a discrete random variable X that can take N different values is given by one of the most famous formulas in information sciences: $H(X) = -\sum_{i=1}^{N} p(x_i) \log_2 p(x_i)$ where $p(X_i)$ is the probability mass associated with the value x_i of X.

- Using the formula shown above, convince yourself of the following: **(1)** When the random variable X takes 8 $(= 2^3)$ different values, each with a probability of $1/8$, the entropy $H(X) = 3$ bits; **(2)** When X takes 256 $(= 2^8)$ different values, each with a probability of $1/256$, we have $H(X) = 8$ bits; and **(3)** If only one specific value is observed for X, $H(X) = 0$. In general, the wider the distribution of the values for a random variable and also when it is more uniform, the greater the entropy. A non-uniform distribution has a lower entropy compared to the uniform case for the same width of the distribution.

Understanding Cross Entropy

- Cross entropy is a measure of the uncertainty that remains in the *predicted* or *estimated* probability distribution for a given random variable vis-a-vis its *true* probability distribution.

- In general, when we use a neural network for image classification, we assume that the classification label for the input image is known precisely. But, just for the sake of presenting a general formula for cross-entropy, let's assume that we are somewhat uncertain about the true identity of the input image and this uncertainty is described by the probability distribution $p_i, i = 0, \ldots, 9$, assuming we are still dealing with the 10-class example mentioned in the previous section.

- At the same time, let $q_i, i = 0, \ldots, 9$ represent the probabilities estimated at the 10 output nodes of the neural network through the ratios shown earlier: $q_i = \frac{e^{x[i]}}{\sum_{j=0}^{9} e^{x[j]}}, \quad i = 0, \ldots, 9$

Revisiting the Cross-Entropy Loss

- In general, if p_i is the probability that the input image belongs to class i and that q_j is the probability associated with the output value at the j^{th} output node through a probabilistic characterization of the output as explained previously, the cross-entropy between the two probability distributions would be given by

$$H_{cross}(p, q) \quad = \quad - \sum_{i=0}^{C-1} p_i \cdot \log_2 q_i \tag{2}$$

  where I have assumed that we have C classes in our training data.

- The summation shown on the right in the definition shown above takes its smallest value when the estimated probabilities q_i's are identical to the true probabilities p_i's, in which case the right hand side shown above yields the entropy for the random variable in question.

Revisiting the Cross-Entropy Loss (contd.)

- Any departure in the estimated q_i values vis-a-vis the true p_i can only increase the value of the cross-entropy as given by Eq.(2).

- For an easy-to-visualize example of the cross-entropy becoming larger than its least possible value mentioned at the bottom of the previous slide, first note that the q_i's must always all add up to 1 since they are after all probabilities.

- Now consider the case with one of the q_i goes to zero (while the value some other q_j acquires the mass that was previously in q_i). Since $\log x \rightarrow -\infty$ as $x \rightarrow 0$, the value of the cross-entropy will become infinity.

Revisiting the Cross-Entropy Loss (contd.)

- Getting back to how the cross-entropy loss is actually used in a network, we always assume that the class label for the input image is known with certainty. If the integer c is the class label for a given image , we assume that $p_j = 1 \; for \; j = c \; and \; 0 \; otherwise$. For such cases, the cross-entropy formula of Eq. (2) becomes:

$$H_{cross}(p, q) \quad = \quad -\log_2 q_c \tag{3}$$

- Recall that if $x[i]$ is the value at the i^{th} node of the output layer of a neural network meant for classification, $q_c = \frac{e^{x[c]}}{\sum_{j=0}^{9} e^{x[j]}}$ for the case when we have 10 classes in our data.

Outline

PyTorch's `torch.nn.CrossEntropyLoss` Class

- The cross-entropy value shown in Eq. (3) on the previous slide is what is measured as the cross entropy loss by a callable instance of the PyTorch class `torch.nn.CrossEntropyLoss` that you can access through the link:

  https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

- As the documentation page says, the `torch.nn.CrossEntropyLoss` function expects to see at output of the final layer of your neural network an **unnormalized scores for each class**. What that translates into is that in your own code you do not have to worry about translating the output in the final layer of your neural network into numbers that look like probability masses; **the `torch.nn.CrossEntropyLoss` class will take care of that for you.**

- **IMPORTANT:** `torch.nn.CrossEntropyLoss` expects that if C is the total number of classes in your training data, your class labels are integers between 0 and $C-1$.

PyTorch's `torch.nn.CrossEntropyLoss` Class (contd.)

- Given the values $x[i], i = 0, \ldots, C-1$, at the nodes of the output layer, for each image in the input batch, CrossEntropyLoss calculates

$$Loss(x, c) = -\log\left(\frac{exp(x[c])}{\sum_{j=0}^{C-1} exp(x[j])}\right) = -x[c] + \log\left(\sum_{j=0}^{C-1} exp(x[j])\right) \tag{4}$$

  where x is the tensor that represents the values in the output layer of the neural network and c is the index value of the true class for the input image.

- `torch.nn.CrossEntropyLoss` returns the average of the loss values for all the images in a batch and that's what is backproped when you call `.backward()` on it.

- A nice thing about the `torch.nn.CrossEntropyLoss` is that it lets you weight the loss calculations to deal with what is referred to as the class imbalance problem in your training data.

PyTorch's `torch.nn.CrossEntropyLoss` Class (contd.)

- One more thing about `torch.nn.CrossEntropyLoss`: This criterion combines the `torch.nn.LogSoftmax` activation function and the `torch.nn.NLLLoss` loss function in one single class. The `LogSoftmax` activation function calculates the log-ratio $\log\left(\frac{exp(x[i])}{\sum_{j=0}^{C-1} exp(x[j])}\right)$ for every node index i in the output layer of the neural network. Subsequently, the `NLLLoss` loss function returns the negative of one of these values that corresponds to the true label of the input image. The name `NLLLoss` stands for "Negative Log Likelihood Loss".

- Most neural networks for image classification consist of convolutional layers followed by a small number of fully connected (FC) layers. The number of nodes in the last FC layer equals the number of image classes in your training data. The comment made above implies that when you use the `torch.nn.CrossEntropyLoss` loss criterion, you do **NOT** need an activation function for the final layer since the `LogSoftmax` activation is built into the loss calculation.

Training vs. Testing "Asymmetry" in Assessing the Output of a Classifier

- As mentioned earlier, a typical CNN for classification consists of several convolutional layers followed by a one or more fully-connected (`torch.nn.Linear`) layers. For example, shown below are the uppermost layers of DLStudio's neural-network class `Net2` for a CIFAR-10 based demo of image classification:

```
class Net2(nn.Module):
    def __init__(self):
        super(DLStudio.ExperimentsWithCIFAR.Net2, self).__init__()
        //  several convolutional layers go here
        self.conv3 = nn.Conv2d(in_ch, out_ch, ker_size, padding=1)
        self.pool3 = nn.MaxPool2d(patch_size, pool_stride)
        // what follows are the fully connected layers:
        in_size_for_fc = out_ch * (32 // np.prod(strides)) ** 2
        self.fc1 = nn.Linear(in_size_for_fc, 150)
        self.fc2 = nn.Linear(150, 100)
        self.fc3 = nn.Linear(100, 10)

    def forward(self, x):
        // the rest of what goes into forward
        // ...
        x = self.pool2(x)
        x = self.pool3(self.relu(self.conv3(x)))
        x = x.view(-1, self.in_size_for_fc)
        x = self.relu(self.fc1( x ))
        x = self.relu(self.fc2( x ))
        x = self.fc3(x)
        return x
```

Training vs. Testing "Asymmetry" ... (contd.)

- As shown on the previous slide, the final layer of this network consists of 10 nodes for the 10 classes in the CIFAR-10 dataset.

- In the training loop shown below, we get those 10 output values for each image in the batch in the call to the network in line (2). With the loss criterion set to `CrossEntropyLoss` in line (1), what's interesting is that at training time we never have to identify explicitly the node that has the predicted label for the input image. That's because the awareness of that node and what to do with it is internal to `CrossEntropyLoss` called in line (3):

```
def run_code_for_training(self, net, display_images=False):
    // ...
    criterion = nn.CrossEntropyLoss()                                                          ## (1)
    for epoch in range(self.epochs):
        for i, data in enumerate(self.train_data_loader):
            inputs, labels = data
            inputs = inputs.to(self.device)
            labels = labels.to(self.device)
            optimizer.zero_grad()
            outputs = net(inputs)                # 'outputs' is  a tensor of 10 elements     ## (2)
            loss = criterion(outputs, labels)    # All 10 values go into the criterion       ## (3)
            running_loss += loss.item()
            //...
```

Training vs. Testing "Asymmetry" ... (contd.)

- However, at test time, we have no choice but to identify the output node that corresponds to the predicted label. At test time, we feed the input batch into the network in line (4), and the output is – as was the case in the training loop – a tensor with 10 elements.

```
def run_code_for_testing(self, net, display_images=False):
    net.load_state_dict(torch.load(self.path_saved_model))
    net = net.eval()
    net = net.to(self.device)
    // ...
    // ...
    with torch.no_grad():
        for i,data in enumerate(self.test_data_loader):
            images, labels = data
            images = images.to(self.device)
            labels = labels.to(self.device)
            outputs = net(images)                  # 'outputs' is  a tensor of 10 elements     ## (4)
            _, predicted = torch.max(outputs.data, 1)                                          ## (5)
            // ...
            // ...
```

- The call to `max()` shown in line (5) returns two things: the max value and its index in the 10 element output vector. We are only interested in the index – since that is the predicted class label.

Binary Cross Entropy Loss

- 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 in Eq.(2) can be expressed as

$$
\begin{aligned}
H_{cross}(p, q) &= -\left[p_0 \cdot \log_2 q(\mathrm{x}[0]) \;+\; p_1 \cdot \log_2 q(\mathrm{x}[1]) \right] \\
&= -\left[p \cdot \log_2 q \;+\; (1 - p) \cdot (1 - q) \right] \qquad (5)
\end{aligned}
$$

- In these equations, the labels for the two classes are denoted '0' and '1'. x[0] and x[1] denote the values at the two output nodes. The quantities $q(\mathrm{x}[0])$ $q(\mathrm{x}[1])$ denote the probabilistic interpretations of the values at the nodes (to be obtained in the same manner as for the multi-class case).

PyTorch's `BCELoss` Cross Entropy Loss

- In the second equation shown on the previous slide, I have denoted p_0 by p, which would make $p_1 = 1 - p$, and $q(x[0])$ by q, which would make $q(x[1]) = 1 - q$.

- 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`

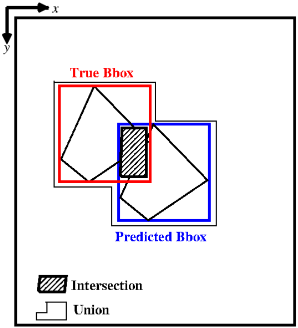
Outline

Measuring the Regression Loss

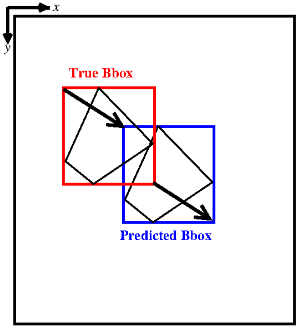
- While the cross-entropy loss takes care of the classification error, it's not appropriate as a measure of the bounding-box regression error.
- Bounding-box regression is about the numerics of where exactly the object is in an image and requires a measure that is more geometrical in nature.
- **Bounding-box 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".

IoU and MSE Losses Illustrated

- You can use the `torch.nn.MSELoss` class for measuring the MSE loss.
- However, you have to program up the IoU loss yourself.



IoU Loss



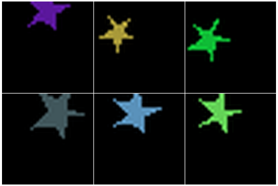
MSE Loss

Outline

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 32×32) (or, perhaps, 64×64 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.**

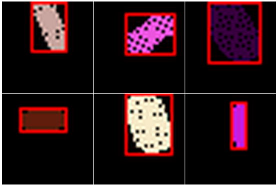
Some Example Images from the PurdueShapes5 Dataset



(a) random stars

(b) with bbox annotations

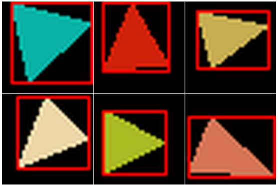
(a) noisy ovals



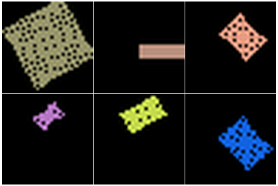
(b) with bbox annotations

Some Example Images from the PurdueShapes5 Dataset (contd.)

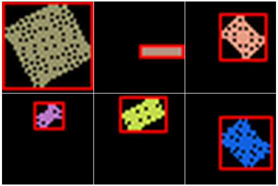
(a) random triangles



(b) with bbox annotations

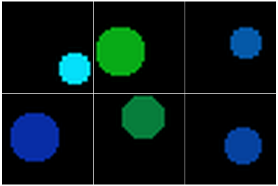


(a) noisy rectangles

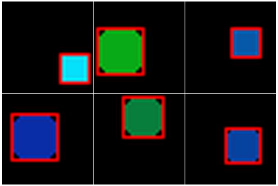


(b) with bbox annotations

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 32×32 image in the dataset is stored using the following format:

```
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:

```
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]
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

Custom Dataloaders and PyTorch

- 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):                              ## must return the size of the dataset
        return len(self.dataset)

    def __getitem__(self, idx):                     ## extracts each image 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 = 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(dataserver_train,
                    batch_size=self.dl_studio.batch_size,shuffle=True, num_workers=4)
    self.test_data_loader = torch.utils.data.DataLoader(dataserver_test,
                    batch_size=self.dl_studio.batch_size,shuffle=False, num_workers=4)
```

Outline

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 the predictions of both the object class label and the 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 object detection and localization with DLStudio.**

The LOADnet2 Network

```
class LOADnet2(nn.Module):
    """
    '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, 5)               ## for the 5 output classes

        ##  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)
        )
        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)        ## output for the 4 coords (x_min,y_min,x_max,y_max) of BBox
        )
```


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

The LOADnet2 Network (contd.)

(...... continued from the previous slide)

```
def forward(self, x):
    x = nn.MaxPool2d(2,2)(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 Bounding Box regression:

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

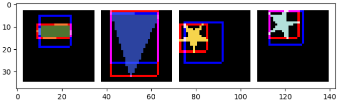
Outline

Training the LOADnet2 Network

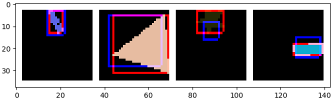
```
def run_code_for_training_with_CrossEntropy_and_MSE_Losses(self, net):
    ...
    net = net.to(self.dl_studio.device)
    criterion1 = nn.CrossEntropyLoss()
    criterion2 = nn.MSELoss()
    optimizer = optim.SGD(net.parameters(), lr=self.dl_studio.learning_rate, momentum=self.dl_studio.momentum)
    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[epoch=%d iter=%d:] Ground Truth:     " % (epoch+1, i+1) +
                ' '.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
            ## code for displaying intermediate results
            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
```

The Two Losses vs. the Iterations During Training

```
[epoch:1/2  iter= 500  elapsed_time=   17 secs]      Ground Truth:           oval   triangle       star       star
[epoch:1/2  iter= 500  elapsed_time=   17 secs]  Predicted Labels:           oval   triangle       star       star
                    gt_bb:  [6,6,21,12]
                  pred_bb:  [7,2,21,16]
                    gt_bb:  [4,0,25,29]
                  pred_bb:  [4,1,25,23]
                    gt_bb:  [0,6,12,18]
                  pred_bb:  [2,5,19,23]
                    gt_bb:  [5,0,18,12]
                  pred_bb:  [5,0,20,15]
[epoch:1/2  iter= 500  elapsed_time=   17 secs]       loss_labelling 1.007        loss_regression: 29.076
```

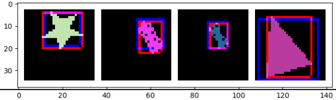


```
[epoch:1/2  iter=1000  elapsed_time=   95 secs]      Ground Truth:      rectangle   triangle       star  rectangle
[epoch:1/2  iter=1000  elapsed_time=   95 secs]  Predicted Labels:           oval   triangle       star       disk
                    gt_bb:  [12,0,18,10]
                  pred_bb:  [11,0,19,11]
                    gt_bb:  [6,2,31,28]
                  pred_bb:  [4,2,28,25]
                    gt_bb:  [9,0,21,10]
                  pred_bb:  [12,5,19,13]
                    gt_bb:  [18,15,31,20]
                  pred_bb:  [19,12,30,2
[epoch:1/2  iter=1000  elapsed_time=                                                        ression: 9.106
```

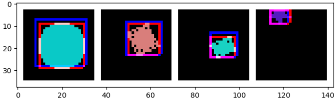


The Two Losses vs. the Iterations During Training (contd.)

```
epoch:1/2  iter=1500  elapsed_time=  156 secs]      Ground Truth:           star       oval  rectangle   triangle
[epoch:1/2  iter=1500  elapsed_time=  156 secs]  Predicted Labels:           star       oval  rectangle   triangle
                    gt_bb:  [8,1,26,17]
                  pred_bb:  [9,1,25,16]
                    gt_bb:  [17,5,27,19]
                  pred_bb:  [16,4,28,17]
                    gt_bb:  [14,6,22,17]
                  pred_bb:  [13,5,22,17]
                    gt_bb:  [5,3,25,30]
                  pred_bb:  [1,4,28,31]
[epoch:1/2  iter=1500  elapsed_time=  1                                                       ression: 5.423
```

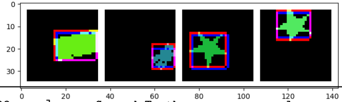



```
[epoch:1/2  iter=2000  elapsed_time=  263 secs]      Ground Truth:           disk  rectangle  rectangle       star
[epoch:1/2  iter=2000  elapsed_time=  263 secs]  Predicted Labels:           disk  rectangle       oval       star
                    gt_bb:  [7,6,27,26]
                  pred_bb:  [5,4,28,25]
                    gt_bb:  [12,6,26,20]
                  pred_bb:  [11,5,27,20]
                    gt_bb:  [14,12,25,21]
                  pred_bb:  [14,10,26,21]
                    gt_bb:  [6,0,15,6]
                  pred_bb:  [6,0,14,6]
[epoch:1/2  iter=2000  elapsed_time=  263 secs]       loss_labelling 0.470       loss_regression: 3.161
```

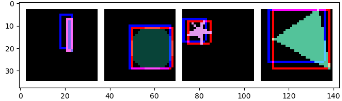


The Two Losses vs. the Iterations During Training (contd.)

```
[epoch:1/2  iter=2500  elapsed_time=  330 secs]      Ground Truth:      rectangle  rectangle        star        star
[epoch:1/2  iter=2500  elapsed_time=  330 secs]   Predicted Labels:          oval        oval        star        star
                   gt_bb:  [12,9,31,22]
                 pred_bb:  [13,10,31,22]
                   gt_bb:  [21,15,31,26]
                 pred_bb:  [21,17,31,25]
                   gt_bb:  [3,10,19,25]
                 pred_bb:  [4,11,20,26]
                   gt_bb:  [7,0,22,13]
                 pred_bb:  [7,1,22,12]
[epoch:1/2  iter=2500  elapsed_time=  330 secs]       loss_labelling 0.448       loss_regression: 2.864
```

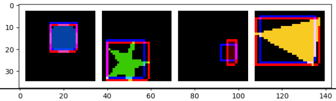



```
[epoch:2/2  iter= 500  elapsed_time=  409 secs]      Ground Truth:          oval        disk        star   triangle
[epoch:2/2  iter= 500  elapsed_time=  409 secs]   Predicted Labels:          oval        disk        star   triangle
                   gt_bb:  [18,4,20,18]
                 pred_bb:  [15,2,20,17]
                   gt_bb:  [12,8,30,26]
                 pred_bb:  [11,7,29,26]
                   gt_bb:  [2,5,12,15]
                 pred_bb:  [0,4,10,14]
                   gt_bb:  [5,0,30,26]
                 pred_bb:  [3,0,28,23]
[epoch:2/2  iter= 500  elapsed_time=  409 secs]       loss_labelling 0.336       loss_regression: 2.196
```

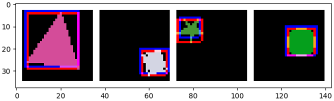


The Two Losses vs. the Iterations During Training (contd.)

```
[epoch:2/2  iter=1000  elapsed_time=  479 secs]      Ground Truth:           disk       star       oval   triangle
[epoch:2/2  iter=1000  elapsed_time=  479 secs]   Predicted Labels:          disk       star       oval   triangle
                    gt_bb:  [11,6,23,18]
                  pred_bb:  [11,5,23,17]
                    gt_bb:  [2,14,21,31]
                  pred_bb:  [2,13,19,30]
                    gt_bb:  [22,13,26,24]
                  pred_bb:  [19,15,26,22]
                    gt_bb:  [0,3,29,24]
                  pred_bb:  [2,2,27,23]
[epoch:2/2  iter=1000  elapsed_time=  479 secs]       loss_labelling 0.300       loss_regression: 2.318
```

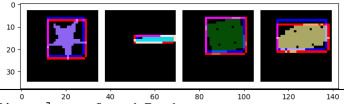



```
[epoch:2/2  iter=1500  elapsed_time=  599 secs]      Ground Truth:       triangle  rectangle       star       disk
[epoch:2/2  iter=1500  elapsed_time=  599 secs]   Predicted Labels:      triangle  rectangle       star       disk
                    gt_bb:  [2,1,25,26]
                  pred_bb:  [1,0,25,25]
                    gt_bb:  [19,18,30,29]
                  pred_bb:  [18,17,31,28]
                    gt_bb:  [0,4,11,14]
                  pred_bb:  [1,3,10,12]
                    gt_bb:  [15,8,27,20]
                  pred_bb:  [14,7,28,20]
[epoch:2/2  iter=1500  elapsed_time=  599 secs]       loss_labelling 0.299       loss_regression: 1.433
```

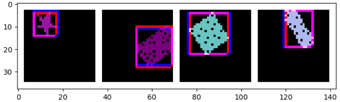


The Two Losses vs. the Iterations During Training (contd.)

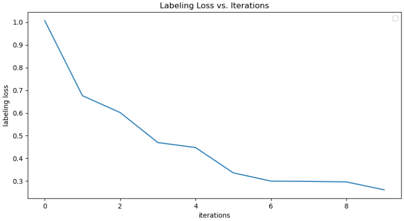
```
[epoch:2/2  iter=2000  elapsed_time=  657 secs]      Ground Truth:           star       oval  rectangle       oval
[epoch:2/2  iter=2000  elapsed_time=  657 secs]   Predicted Labels:          star       oval       disk       oval
                    gt_bb:   [9,4,26,21]
                  pred_bb:   [9,3,25,20]
                    gt_bb:   [13,11,31,14]
                  pred_bb:   [14,11,29,13]
                    gt_bb:   [10,3,28,19]
                  pred_bb:   [10,3,27,18]
                    gt_bb:   [7,6,30,18]
                  pred_bb:   [7,4,29,17]
[epoch:2/2  iter=2000  elapsed_time=  657 secs]       loss_labelling 0.297       loss_regression: 1.266
```



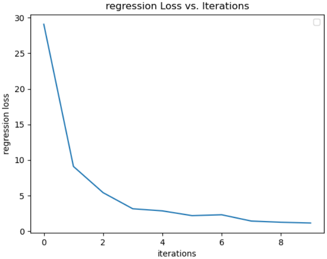

```
[epoch:2/2  iter=2500  elapsed_time=  754 secs]      Ground Truth:           star  rectangle  rectangle       oval
[epoch:2/2  iter=2500  elapsed_time=  754 secs]   Predicted Labels:          star       oval  rectangle       oval
                    gt_bb:   [4,1,14,11]
                  pred_bb:   [4,0,15,11]
                    gt_bb:   [15,7,31,25]
                  pred_bb:   [15,8,31,24]
                    gt_bb:   [4,1,21,19]
                  pred_bb:   [3,1,22,19]
                    gt_bb:   [12,0,24,16]
                  pred_bb:   [11,0,24,16]
[epoch:2/2  iter=2500  elapsed_time=  754 secs]       loss_labelling 0.262       loss_regression: 1.158
```



Training and Regression Losses versus Iterations



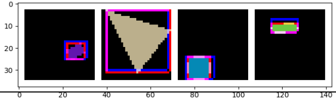
(a) labeling loss vs. iterations



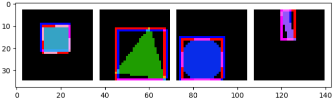
(b) regression loss vs. iterations

Results on Unseen Test Data

```
[i=0:] Ground Truth:      rectangle   triangle        disk       oval
[i=0:] Predicted Labels:  rectangle   triangle        disk  rectangle
                   gt_bb:   [19,15,27,22]
                 pred_bb:   [18,14,28,22]
                   gt_bb:   [2,0,31,28]
                 pred_bb:   [2,0,30,27]
                   gt_bb:   [4,21,14,31]
                 pred_bb:   [3,21,16,31]
                   gt_bb:   [7,6,19,10]
                 pred_bb:   [7,4,19,10]
```



```
[i=100:] Ground Truth:      rectangle   triangle        disk       oval
[i=100:] Predicted Labels:  rectangle       disk        disk       oval
                   gt_bb:   [9,7,21,19]
                 pred_bb:   [8,6,21,18]
                   gt_bb:   [7,8,29,31]
                 pred_bb:   [8,9,30,31]
                   gt_bb:   [2,13,20,31]
                 pred_bb:   [1,12,21,31]
                   gt_bb:   [12,0,18,13]
                 pred_bb:   [12,0,16,13]
```



Classification Accuracy on the Unseen Test Data (After 2 Epochs of Training)

```
Prediction accuracy for rectangle : 64 %
Prediction accuracy for triangle : 97 %
Prediction accuracy for  disk : 99 %
Prediction accuracy for  oval : 81 %
Prediction accuracy for  star : 99 %



Overall accuracy of the network on the 1000 test images: 88 %


Displaying the confusion matrix:
```

	rectangle	triangle	disk	oval	star
rectangle:	64.00	0.50	1.00	31.50	3.00
triangle:	1.50	97.50	1.00	0.00	0.00
disk:	1.00	0.00	99.00	0.00	0.00
oval:	18.50	0.00	0.50	81.00	0.00
star:	0.00	0.50	0.00	0.00	99.50