# Multi-Instance Object Detection – Image Cells and Anchor Boxes

**Avinash Kak**

**Purdue University**

**Lecture Notes on Deep Learning**
**Avi Kak and Charles Bouman**

Tuesday 25th February, 2025 10:17

# 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 the first part of this lecture, I'll 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.**

# Preamble (contd.)

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.

With regard to 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 the first part 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.

The first part of this lecture will also introduce you to a new dataset, `PurdueShapes5`, of $32 \times 32$ 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.

# Preamble (contd.)

As you would expect, any new dataset for training a CNN calls for a custom dataloader. DLStudio includes a dataloader for the `PurdueShapes5` dataset.

The second part of this lecture deals with the more difficult problem of detecting multiple objects simultaneously in a single image. Before I delve into how one can solve such a problem with a neural network, allow me to make some general comments about the level of difficulty involved in programming a neural network.

In general, the simpler the relationship between the input to a neural network and its expected output, the easier it is to program the neural network. The problem of image classification represents the easiest end of the spectrum of difficulty levels of programming neural networks. Here the input/output relationship is as simple as it can be – the input is the entire image and the output its corresponding label.

The problem of object detection and localization under the assumption that the image contains a single object instance would be at the next level of difficulty. Now the input/output relationship is more complex since, in addition to a prediction for the class label, the output must also include predictions for the four numerical parameters that localize the bounding box for the object.

# Preamble (contd.)

For an even more challenging problem, consider the case when the images are allowed to contain multiple object instances and your job is both to identify the instances and to localize them individually. The input/output relationship in this case is substantially more complex and, unless you have previously seen someone else's solution for the problem, it would not be immediately obvious as to how one would go about solving the problem with a neural network.

Of the various deep-learning based solutions that have been proposed for solving the problem of multi-instance detection and localization, the following three are the most prominent:

R-CNN In this approach an RPN (Region Proposal Network) is used to first propose different possible bounding boxes for the objects of interest in an image. Subsequently, a classifier network is applied to the pixels in each of the proposed bounding boxes. Finally, there is a post-processing step to refine the bounding boxes associated with the classifications that carry the highest confidence values and for eliminating duplicate detections through overlapping bounding boxes.

# Preamble (contd.)

As for the name of this approach, the letter 'R' in R-CNN stands for "Regions" for the output of the Region Proposal Network. R-CNN is described in the following publication:

https://arxiv.org/pdf/1311.2524.pdf

YOLO What makes R-CNN relatively complex is that it consists of multiple neural networks (RPN and the classifier) that must be trained and fine-tuned separately. And then there is also the issue of the post-processing of the results as mentioned above. The YOLO approach, on the other hand, consists of using a single neural network. The image is divided into a grid of cells and the job of predicting the bounding-box and the class-label for an object in the image is the responsibility of the grid cell that has the center of the object in it. Here is the link to the original YOLO paper:

https://arxiv.org/pdf/1506.02640.pdf

# Preamble (contd.)

There now exist several versions of YOLO, with some of the more recent versions meant for multi-instance object detection in videos and doing so in real time. Here's a link to a 2024 publications that presents a real-time version of YOLO:

https://arxiv.org/pdf/2405.14458

The acronym YOLO stands for "You Only Look Once" that's meant to contrast this approach with that of R-CNN that requires more than one neural-network to process an input image.

SSD The name stands for "Single Shot Detector" — the name is meant to convey the fact that, as in YOLO, this approach also processes an input image only once. SSD is based on defining a set of default bounding boxes in the feature maps produced by the convolutional layers. The output of each convolutional layer provides predictions for the offsets associated with the true bounding boxes from the default bounding boxes and for the class labels to be associated with the offset bounding boxes.

# Preamble (contd.)

Each feature layer directly informs the output about its estimated probabilities related to the bounding-box offsets and class labels. Here is the link to the original paper on SSD:

https://arxiv.org/pdf/1512.02325.pdf

In the second part of this lecture, I'll focus on the architectural details of the YOLO framework. The computational steps that implement these details are programmed into the following version of my YOLOLogic Python module:

https://engineering.purdue.edu/kak/distYOLO/

My YOLO presentation will explain how the multi-instance object detection in images is facilitated by first overlaying a grid of cells on the input image and then defining a certain number, five to be exact in our case, of anchor boxes for each cell of the grid. Each anchor-box is characterized by its aspect ratio (height to width ratio). Subsequently, during training the YOLO network, you assign each object in the input image to that cell for which the center of the object is closest to the center of the cell. Going further, within each cell, you assign the object to that anchor box whose aspect ratio is closest to that of the object.

# Preamble – How to Learn from These Slides

Since it is a large slide deck, you may need some help with how to digest all the information that is presented here.

To that end, note that the most fundamental ideas covered in this lecture are just the following three:

- Cross entropy for measuring loss. To understand cross-entropy you must possess a good understanding of the notion of entropy. **The concepts of entropy and cross-entropy are covered in Slide 19 through 37**.
- How to measure regression losses in a network. **This topic is discussed in Slides 38 through 49**.
- The basic ideas of the YOLO logic for Multi-instance object detection. **These are presented in Slides 94 through 109**.

That makes for a total of just 44 slides you need to focus on at the beginning. Only after you have understood the material in these 44 slides, you should take the time to look over what's in the rest of the slides. Most of that material covers network details, the dataset attributes, the results, etc.

# 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.

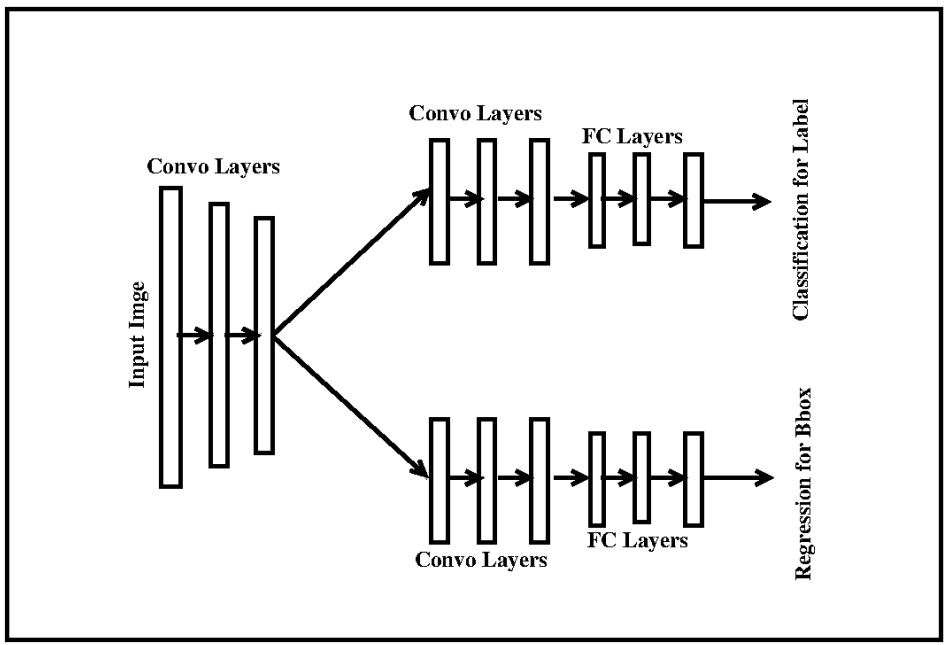



Figure: A dual-inferencing CNN

# Now We Need Two Different Loss Functions

- In the homework you just submitted, you used the cross-entropy loss to measure the prediction errors in a purely classification network. For that you used the PyTorch class `torch.nn.CrossEntropyLoss`.
- In this lecture, I'll start by giving you a deeper understanding of Cross Entropy Loss. As you will see, it is one of the most beautiful concepts in neural learning.
- Subsequently, I will argue that while cross-entropy is great as a measure of the misclassification error, it does NOT have the right semantics for what's needed for the regression error that measures the accuracy in estimating the coordinates of the bounding box for the object of interest.
- 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.

# Cross-Entropy Loss for Measuring Classification Errors

- For solving the classification problem mentioned in the last bullet of the previous slide, let **x** represent the input image and let the vector **y** represent the values at the 10 output nodes. We assume that the true class label for the input **x** is $c$, which is an integer between 0 and 9. The cross-entropy loss for this output would be given by

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

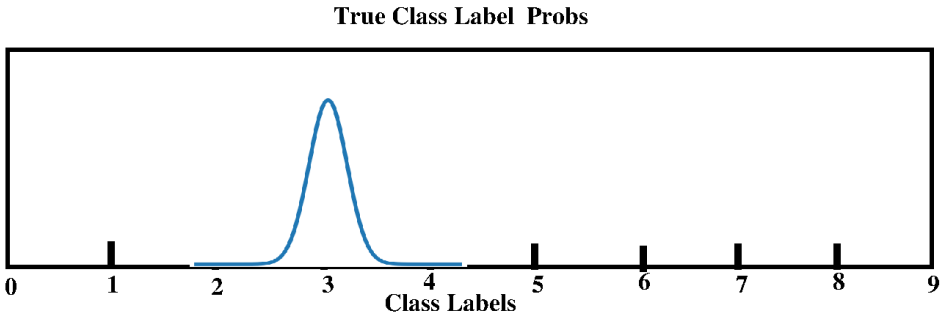
- For the numerator in the ratio shown, you sample the output layer only at that node whose index corresponds to the true class label of the input image.

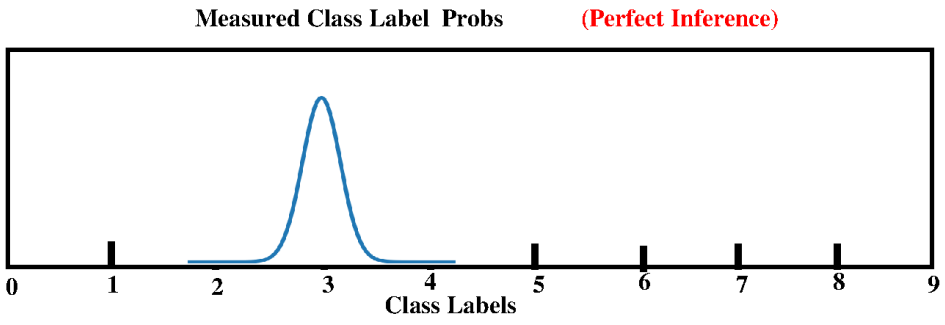
- To see why the formula shown above makes total sense for measuring the classification loss when the predicted label is wrong for a given input, first focus on the fact that, if the inferencing were to be perfect, the output element $\mathbf{y}[c]$ for the particular index $c$ in question would light up the strongest in relation to all the other elements in the vector **y**.

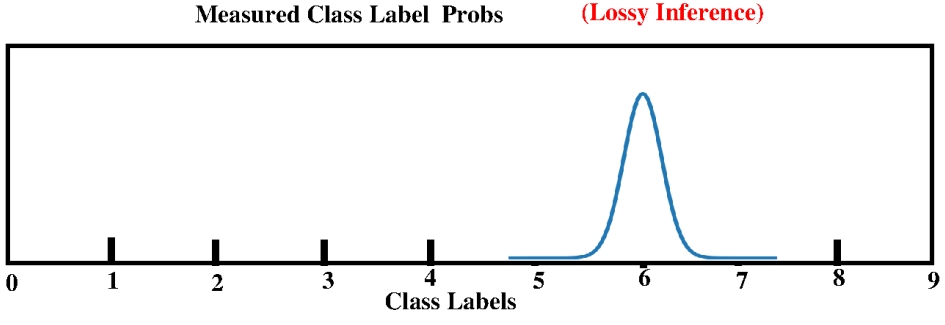
## Cross-Entropy Loss for Measuring Classification Errors (contd.)

- Assuming $c = 3$, perfect inference is illustrated by the middle plot on the next slide. The top plot in the slide shows the probabilities associated with the different possible class labels for the input image.
- For perfect inference, both the numerator and the denominator in the formula on the previous slide would be *roughly* equal to 1. And since the log of 1 is zero, the loss in this case would be close to zero.
  [**Assume that the final activation function yields floating-point values over the full $(-1.0, 1.0)$ interval. For perfect inference, for $c = 3$, assume that $\mathbf{y}[3] = +1.0$ and that $\mathbf{y}[c] = -1.0, c \neq 3$. In this case, the numerator of just the ratio part of the expression shown in Eq. (1) would be $e$ and the denominator would be $e + 9/e$.**]
- The bottom plot on the next slide shows an example of the case when the predicted class label is wrong. Now the output node that lights up the strongest does not correspond to the true class label of the input image. **But you still sample the output layer at the node whose index corresponds to the true class label of the input image.** In our example, that is $c = 3$.
- When the predicted class label is wrong, the numerator in the ratio part of the formula would evaluate to a small value. **However, the summation in the denominator is likely to evaluate to roughly the same value**.

# Cross-Entropy Loss for Measuring Classification Errors (contd.)



## Cross-Entropy Loss for Measuring Classification Errors (contd.)

- Therefore, when the predicted class label is wrong, the ratio in the formula on Slide 14 would be a small fractional number **whose logarithm would be a large negative number**. **The formula would therefore return a large positive number for loss**.
- When the predicted class label is wrong, that's commonly referred to as **lossy inference.**
- Note that, for at least the example illustrated by the plots in the previous slide, for both the perfect inference and the lossy inference, **the denominator in the formula on Slide 14 would add up to more-or-less the same value**. **However, the numerator would change significantly from error-free inference to lossy inference.**
- As it turns out, the loss function shown on Slide 14 has deep roots in information sciences — roots that are based on a probabilistic interpretation of that formula.

## Cross-Entropy Loss for Measuring Classification Errors (contd.)

- **VERY IMPORTANT:** The 10 output values given by the ratios $\frac{e^{\mathbf{y}[i]}}{\sum_{j=0}^{9} e^{\mathbf{y}[j]}}$ 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 $\mathbf{y}[i]$ and because these 10 numbers add up to 1.0.

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

# 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: $q_i = \frac{e^{\mathbf{y}[i]}}{\sum_{j=0}^{9} e^{\mathbf{y}[j]}}, \quad i = 0, \ldots, 9$

# 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. [For a proof of this assertion, see Slides 18-20 of my Week 11 Lecture on Adversarial Learning.]

# 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 when, for an index $i$ for which $p_i$ is non-zero, one of the $q_i$'s goes to zero (while the value of 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.

- The difference between the cross-entropy $H_{cross}(p, q)$ and its least value given by the entropy $H_p$ is known as the very famous KL-Divergence that we will talk about in my Week 11 lecture.

# 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 $y[i]$ is the value at the $i^{th}$ node of the output layer of a neural network meant for classification and $c$ is the numeric ground-truth class label of the input image, $q_c = \frac{e^{\mathbf{y}[c]}}{\sum_{j=0}^{9} e^{\mathbf{y}[j]}}$ for the case when we have 10 classes in our data.

# Do Not Confuse Cross-Entropy with Conditional Entropy

- To explain Conditional Entropy: Let's say you have two different random variables, $X$ and $Y$, that are interdependent in some manner. We can talk about a joint probability distribution $p(x, y)$ over the two variables, with $x$ being the realizations of $X$ and $y$ those for $Y$. And we can express the uncertainty associated with the joint distribution by the joint entropy $H(X, Y) = -\sum_x \sum_y p(x, y) \cdot \log p(x, y)$.
- We can now pose the following question: How much uncertainty would remain in, say, $X$ if we had knowledge of the values taken on by $Y$? The answer to this question is supplied by the conditional entropy $H(X|Y)$. One can show that $H(X|Y) = H(X, Y) - H(X)$. [For proof, see pages 13 through 16 of my Decision Tree tutorial at https://engineering.purdue.edu/kak/Tutorials/DecisionTreeClassifiers.pdf]
- Cross Entropy, on the other hand, is about there being two different distributions, $p(x)$ and $q(x)$, for the same random variable $X$. We may consider $p(x)$ to be the true distribution and $q(x)$ an estimate for $p(x)$. Cross Entropy answers the question of how much uncertainty there exists in $q(x)$ vis-a-vis the uncertainty expressed by $p(x)$.

# Outline

# PyTorch's `torch.nn.CrossEntropyLoss` Class

- The cross-entropy value shown in Eq. (3) on Slide 24 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 $\mathbf{y}[i], i = 0, \ldots, C-1$, at the nodes of the output layer, for each image $\mathbf{x}$ in the input batch, `CrossEntropyLoss` calculates

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

  where $\mathbf{y}$ 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 $\mathbf{x}$.

- 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.

- A very important thing about `torch.nn.CrossEntropyLoss`: This criterion combines the `torch.nn.LogSoftmax` activation function and the `torch.nn.NLLLoss` loss function in one single operation.

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

- About the point made in the last bullet of the previous slide: The `LogSoftmax` activation function calculates the log-ratio $\log\left(\frac{e^{\mathbf{y}[i]}}{\sum_{j=0}^{C-1} e^{\mathbf{y}[j]}}\right)$ 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. Inference "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. Inference "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 (B). With the loss criterion set to `CrossEntropyLoss` in line (A), at training time you must also supply the true class label (as an integer index) for each image in the batch. This is accomplished by the statement in Line (C). The second argument to the loss function is the integer index for the true class label of each batch instance.

```
def run_code_for_training(self, net, display_images=False):
    // ...
    criterion = nn.CrossEntropyLoss()                                          ## (A)
    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' shape: (B,10) for 10 classes    ## (B)
            loss = criterion(outputs, labels)      #  All 10 values go into the criterion     ## (C)
            running_loss += loss.item()
            // ...
```

## Training vs. Inference "Asymmetry" ... (contd.)

- At inference time, as at training time, each node at the output corresponds to a class label. At training time, the values at the output nodes are translated into probabilities by the `LogSoftmax` activation which is internal to `CrossEntropyLoss`. In light of those semantics associated with the node values, at inference time, it is sufficient if we merely check as to which node has the highest value.

```
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' shape: (.,10) for 10 classes    ## (D)
            _, predicted = torch.max(outputs.data, 1)                                            ## (E)
            // ...
```

- The call to `max()` shown in line (E) 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 – Theory

- 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) on Slide 22 can be expressed as

$$\begin{aligned} H_{cross}(p,q) &= -\Big[p_0 \cdot \log_2 q(\mathbf{y}[0]) \;+\; p_1 \cdot \log_2 q(\mathbf{y}[1])\Big] \\ &= -\Big[p \cdot \log_2 q \;+\; (1-p)\cdot \log_2(1-q)\Big] \end{aligned} \tag{5}$$

- In the first equation shown above, $p_0$ and $p_1$ are the probabilities associated with the input image that it either belongs to class '0' or to class '1' and $q_0$ and $q_1$ the two output probabilities for the same input image. Note that $\mathbf{y}[0]$ and $\mathbf{y}[1]$ denote the values at the two output nodes. The quantities $q(\mathbf{y}[0])$ and $q(\mathbf{y}[1])$ denote the probabilistic interpretations of the values at the nodes.

# Binary Cross Entropy Loss (contd.)

- The second equation shown on the previous slide simplifies the notation by taking advantage of the fact that $p_0 + p_1 = 1$ and $q_0 + q_1 = 1$. So if we represent $p_0$ by just $p$, we have $p_1 = 1 - p$. Similarly, if we represent $q_0$ by $q$, we have $q_1 = 1 - q$.
- 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 $-\log_2(1 - q)$.

# Binary Cross Entropy Loss – In Practice

- In practice, Binary Cross-Entropy Loss (BCELoss) is used for solving the one-verses-the-rest classification or detection problems. Let's say we are scraping the web and our goal is to collect just the face images. In this case, you would want to incorporate a good face detector in the web scraping algorithm.
- A neural network for such a detector is likely to have a single output node in the final layer. The value at this node would be construed as the value of $q$ in Eq. (5). The value at the other node implied by a literal interpretation of Eq. (5) can be treated implicitly because of the normalization constraint.
- The value you find at the solitary output node in such a network is the output produced by the `nn.Sigmoid()` activation function. That guarantees that the final output value will be between 0 and 1.0 — a necessary condition on a numeric value if it is to be interpreted as a probability. Ordinarily, the numeric values in a neural network range between -1.0 and 1.0.

# Binary Cross Entropy Loss – In Practice (contd.)

- PyTorch provides the class `nn.BCELoss` "Binary Cross Entropy Loss" for the sort of applications mentioned on the previous slide. Here is the documentation page for this loss function:

  `https://pytorch.org/docs/stable/nn.html#bceloss`

- When using `nn.BCELoss` class as a loss criterion, for the target we typically use 1.0 since the goal is to maximize the probability of correct detection of the object of interest. That is, when the input image is of the desired class, we want it to be detected with a probability that is as close as possible to 1.0.

- An example of such a network that produces a single-node output is shown on Slide 71 of my Week 11 slides. Using 1.0 as the target for the loss criterion is shown on Slide 76 (of the same slide deck) in the (A) section of the code.

# Binary Cross Entropy Loss – In Practice (contd.)

- While you are likely to use the target of 1.0 in most cases, PyTorch allows you to use any value for the target. As you will see later in this class in Week 11, I'll present an example that requires us to use 0 as the target. That is, instead of maximizing the probability of detection of a category of objects, we would want to minimize it. [Yes, it does sound weird that anyone would want to do that. You'll just have to wait until the Week 11 lecture to see why that would be a desirable thing to do in some cases.]

- The value you supply for the criterion target is the value for $p$ in the formula shown in Eq. (5) on Slide 33. When the target is $p = 1$, the loss is equal to $-\log_2 q$. On the other hand, if the criterion target is $p = 0$ the loss becomes $-\log_2(1 - q)$. Recall, $q$ is the value at the output of the `nn.Sigmoid` activation at the output node.

# 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. **The loss functions that are used to measure the precision with which a numerical attribute is predicted by a neural network are generally known as regression losses.**
- $L_1$ and $L_2$ norms — the Mean Absolute Error and the Mean Squared Error — are probably the most commonly used loss functions for solving general regression problems. PyTorch gives you `torch.nn.L1Loss` and `torch.nn.MSELoss` for these two regression losses.
- How exactly you use the $L_1$ or $L_2$ norms in your code depends on how you represent the bounding box (BB) that localizes the object you are trying to detect. You have two different choices as shown on the next slide.

# Representing the Bounding Box

- Generally speaking, for representing a bounding box (BB), you have the following two choices:
  - by the 4-tuple $(i_1, j_1, i_2, j_2)$ in which $(i_1, j_1)$ are the coordinates of the upper left corner and $(i_2, j_2)$ the coordinates of the lower right corners; and
  - by the 4-tuple $(i_c, j_c, w, h)$ in which $(i_c, j_c)$ are the coordinates of the center of the BB and $(w, h)$ the its width and height .
- On a per-instance basis in a batch, the $L_2$ loss for the former representation would be $(i_1 - \hat{i}_1)^2 + (j_1 - \hat{j}_1)^2 + (i_2 - \hat{i}_2)^2 + (j_2 - \hat{j}_2)^2$ where the hatted quantities are the predicted values. You can write a similar expression for the $L_1$ loss except that now you would be calculating the absolute differences of the true and the predicted coordinates.
- I have used the $(i_1, j_1, i_2, j_2)$ representation for the results I'll be showing on single-object detection. For multi-object detection later in this lecture, I'll be using the $(i_c, j_c, w, h)$ representation.

# Measuring Regression Loss with `torch.nn.MSELoss`

- Elaborating a bit on the `nn.MSELoss` for the $L_2$ norm — since it is used rather frequently for regression — in keeping with the explanation of the previous slide,

$$nn.MSELoss = \frac{1}{B}\sum_{k=1}^{B}\frac{1}{4}\left[(i_1^k - \hat{i}_1^k)^2 + (j_1^k - \hat{j}_1^k)^2 + (i_2^k - \hat{i}_2^k)^2 + (j_2^k - \hat{j}_2^k)^2\right] \tag{6}$$

  where $B$ is the batch size and where, again, the hatted quantities are the predicted values.

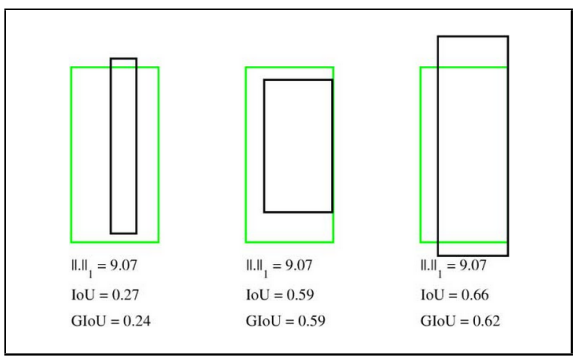
- It is interesting to note that if $(i_c, j_c)$ denotes the center of the ground-truth BB and $(\hat{i}_c, \hat{j}_c)$ the center of the predicted BB, the above formula is used as an approximation to

$$nn.MSELoss = \frac{1}{B}\sum_{k=1}^{B}\frac{1}{2}\left[(i_c^k - \hat{i}_c^k)^2 + (j_c^k - \hat{j}_c^k)^2\right] \tag{7}$$

  That is, we assume that the MSELoss measures the mean-squared-error between the centers of the two BBs in each instance.

# Shortcomings of $L_1$ and $L_2$ Norms for BB Localization Loss

- In what's shown below, the ground-truth BB are in green and the predicted BB in black.
- The depiction at left is based on using the $(i_1, j_1, i_2, j_2)$ BB representation. In all three cases at left, the $L_1$ loss (shown as 9.07) is exactly the same although the predicted BB are significantly different. What's happening is that the differences in the values of corresponding coordinates are getting distributed between the two corners in such a way that does not change the $L_1$ norm.
- Shown at right is a similar result with the $(i_c, j_c, w, h)$ representation and $L_2$ norm (value 8.41).



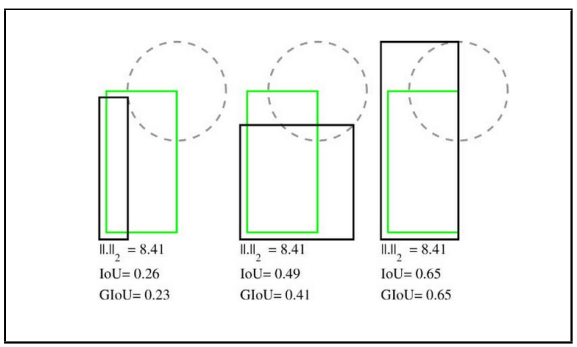
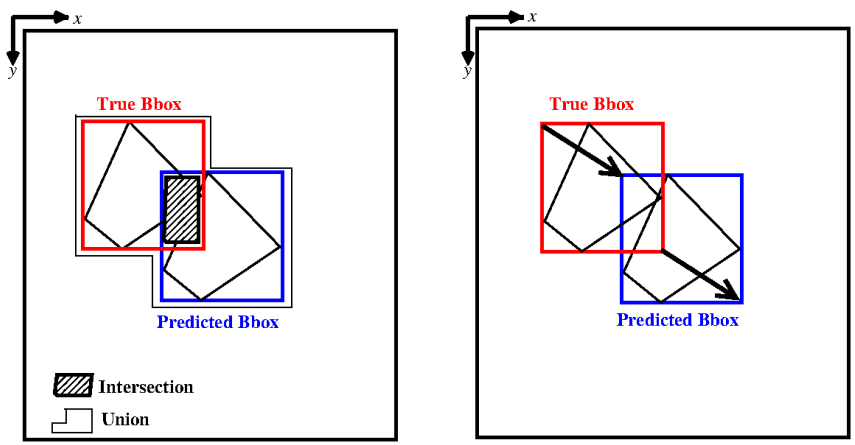




Figure: *These two figures are from* `https://giou.stanford.edu/`

# Loss Based on Intersection-over-Union (IoU)

- What's shown at left below is just the basic idea of IoU (Intersection over Union). As the name implies, you measure the similarity between a ground-truth BB and the predicted BB by taking the ratio of the intersection of the two BBs to their union. You can turn the IoU value thus obtained into a loss by measuring $1 - IoU$.
- However, $1 - IoU$ as a measure of loss has a problem: If the predicted BB does not overlap with the ground-truth BB, the value of loss would remain the same regardless of how far or close the prediction is to the ground-truth. Therefore, its derivatives would be zero —- implying that you would not be able to use it for training a network.

[**Although not useful for training, the *IoU* metric has been used extensively for evaluating detection networks.**]



# Variants of IoU

- The following two publications have proposed variants of the IoU based loss that can be used for training an object detection network:

  https://arxiv.org/pdf/1902.09630.pdf

  https://arxiv.org/pdf/1911.08287.pdf

- First came the Generalized IoU (GIoU) by Rezatofighi et al. in the first paper cited above. And then came the Distance-IoU (DIoU) and Complete-IoU (CIoU) in the second paper by Zheng et al.

- As illustrated in Slide 46, GIoU loss is based on a convex hull $C$ of both the ground-truth BB $B^{gt}$ and the predicted BB $B$ through the following formula that uses the set-theoretic notation '$|.|$' for the cardinality of a set and where the operators '$\cup$' and '$-$' stand for the union and the difference of two sets:

$$GIoU\ Loss \;=\; 1 \;-\; IoU \;+\; \frac{|C - B \cup B^{gt}|}{|C|} \tag{8}$$

# Variants of IoU (contd.)

- The good thing about GIoU is that it creates a differentiable loss function even when the predicted BB has no intersection with the ground-truth BB.
- For large distances between $B$ and $B^{gt}$ — at distances when there is no overlap between them — the value of $IoU$ will stay at zero, and the third term in the loss shown on the previous slide will approach 1.0 since $B \cup B^{gt}$ will be a small fraction of $C$. In such cases, for increasingly larger distances between between $B$ and $B^{gt}$, the loss will approach 2.0. On the other hand, when $B$ and $B^{gt}$ are completely overlapping, the third term will be zero and the loss will be $1 - IoU$.
- But there's a price to pay: For positions of the predicted BB inside the ground-truth BB, GIoU becomes IoU since $B \cup B^{gt} = B^{gt} = C$. Additionally, when $B$ is completely inside $B^{gt}$, the IoU based loss does not change if the only change is in the position of the predicted BB and not its shape. This can slow down the convergence of any learning with GIoU.

# Variants of IoU (contd.)

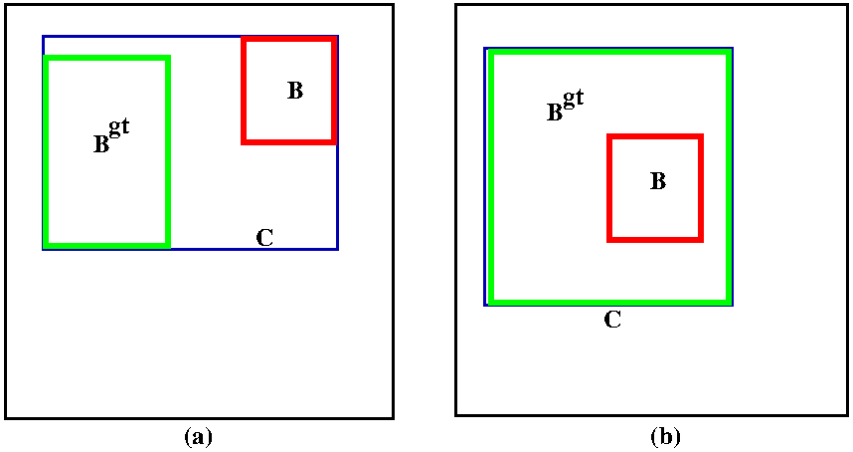



Figure: *GIoU: Shows the relationship between the convex hull C and the bounding boxes $B^{gt}$ for the ground-truth and B for the predicted. (a) No overlap case; and (b) complete overlap case.*

- To remedy the shortcomings of GIoU, the authors of the second paper cited on Slide 44 proposed a couple of "extensions" to GIoU: Distance-IoU (DIoU) and Complete-IoU (CIoU). Here is the formula for the DIoU Loss:

$$DIoU\ Loss = 1 - IoU + \frac{|C - B \cup B^{gt}|}{|C|} + \frac{d^2}{c^2} \tag{9}$$

where $d$ is the distance between the centers of $B^{gt}$ and the predicted $B$, $c$ the length of the diagonal of the convex hull of the two boxes, as shown on the next slide.

# Variants of IoU (contd.)

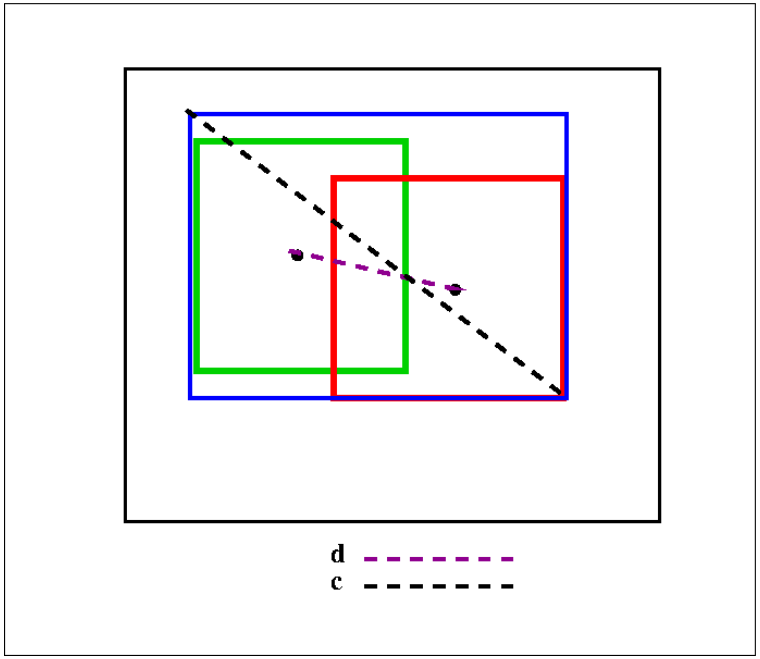



Figure: *DIoU: The green box is the ground-truth bounding box $B^{gt}$, the red box is the predicted $B$; $d$ is the distance between the centers of the two boxes; and $C$ the diagonal of the convex hull. The role of $c$ is to normalize $d$*

- In general, DIoU loss converges much faster than the GIoU loss.
- The authors of DIoU have gone further and claimed that, for best results, one should include in the loss a term that directly compares the aspect ratio of the ground-truth BB and the predicted BB. They called the resulting loss the CIoU for Complete-IoU.

# Variants of IoU (contd.)

- Shown below is the formula for the CIoU Loss:

$$CIoU\ Loss \;=\; 1 \;-\; IoU \;+\; \frac{|C - B \cup B^{gt}|}{|C|} \;+\; \frac{d^2}{c^2} \;+\; \alpha \cdot v \tag{10}$$

- The term $v$ at the end of the RHS in the above equation is for comparing the aspect ratio of the predicted BB vis-a-vis the ground-truth BB and $\alpha$ is a "positive trade-off parameter". First, here is the definition for $v$:

$$v \;=\; \frac{4}{\pi^2}\left(\arctan\frac{w^{gt}}{h^{gt}} \;-\; \arctan\frac{w}{h}\right)^2 \tag{11}$$

  where $w$ and $h$ are the width and the height for the predicted BB and $w^{gt}$ and $h^{gt}$ the same for the ground-truth BB.

- The authors suggest the following value for the trade-off parameter:

$$\alpha \;=\; \frac{v}{(1 - IoU) + v} \tag{12}$$

- Finally, PyTorch provides the following function that you can call for any of the IoU variants in your own code:

https://pytorch.org/vision/stable/generated/torchvision.ops.complete_box_iou_loss.html

# Implementation of IoU Variants in DLStudio

- The inner class `DetectAndLocalize` of DLStudio contains a custom loss function provided by the class `DIoULoss` that provides an implementation for the IoU-based losses defined in Eqs. (8) and (9) in the preceding slides. [It would be straightforward to include additional code in the `DIoULoss` class for the loss function in Eq. (10).]

- In order to experiment with these additional loss functions, the inner class `DetectAndLocalize` also contains the training function `run_code_for_training_with_iou_regression()`. This function requires you to supply an argument for a parameter named `loss_mode` for which of the IoU variants you want to use for the loss function.
[**The possible choices for the `loss_mode` parameter are `d2`, `diou1`, `diou2`, and `diou3`. The first of these does the same thing as the `nn.MSELoss`, the second just adds IoU loss to the first case, the third is an implementation of the loss in Eq. (8), and the last an implementation of the loss in Eq. (9).** ]

- See the following script in the Examples directory of DLStudio:

```
object_detection_and_localization_iou.py
```

for a demonstration of how to use the IoU-variant loss functions in DLStudio.

# Outline

# The PurdueShapes5 Dataset with Bbox Annotations

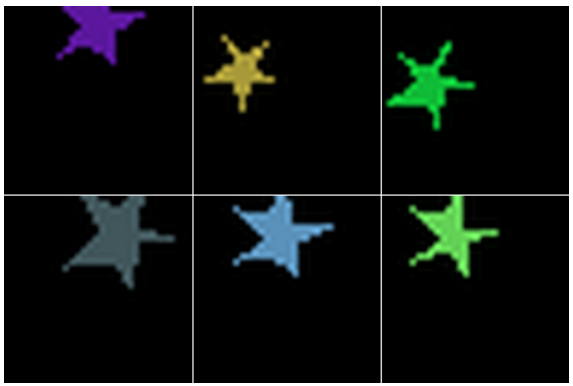
- This dataset of synthetically generated annotated images is used by the following scripts in the Examples directory of DLStudio:

```
object_detection_and_localization.py
noisy_object_detection_and_localization.py
object_detection_and_localization_iou.py
```

- About the motivation that led to the creation of the `PurdueShapes5` dataset, 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 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.**

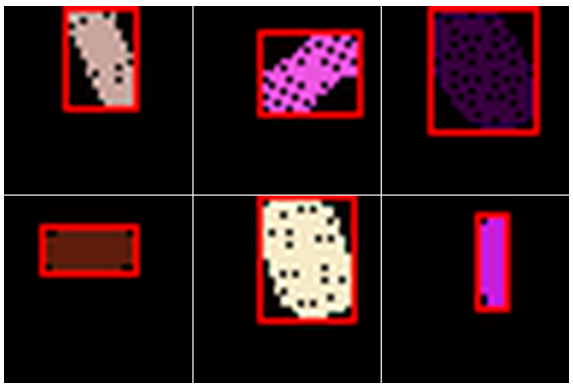
# Some Example Images from the PurdueShapes5 Dataset



(a) random stars

(b) with bbox annotations

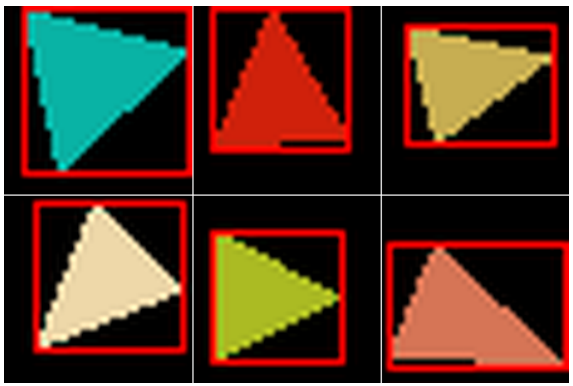
(a) noisy ovals



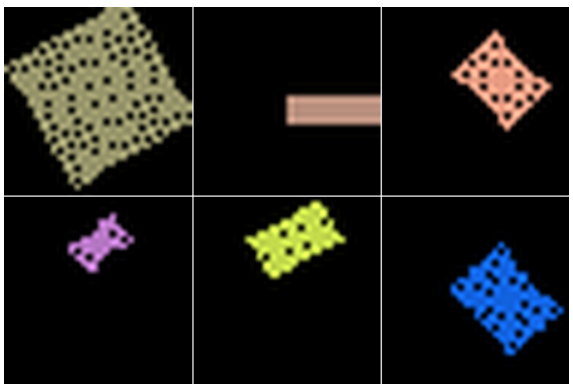
(b) with bbox annotations

# Some Example Images from the PurdueShapes5 Dataset (contd.)

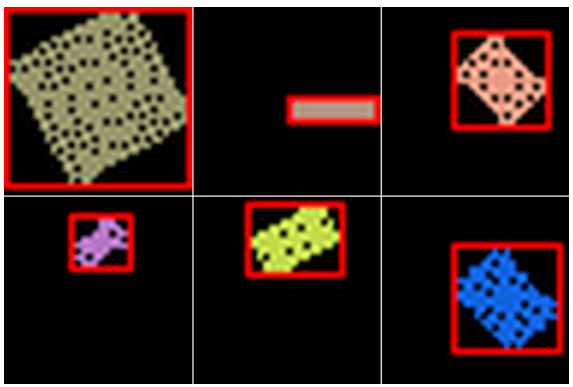
(a) random triangles



(b) with bbox annotations

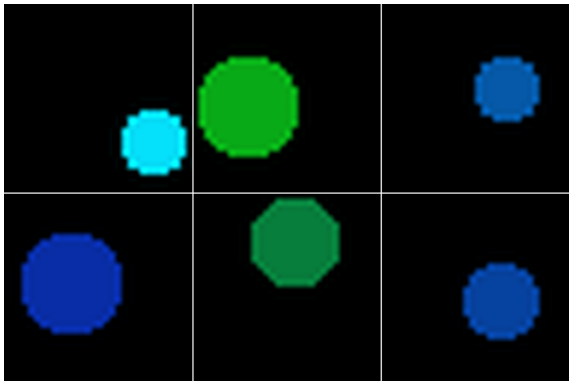


(a) noisy rectangles

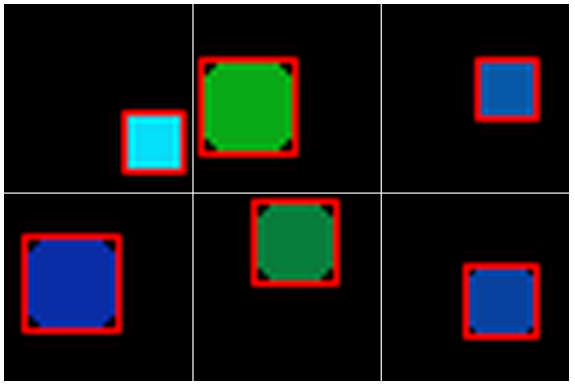


(b) with bbox annotations

# Some Example Images from the PurdueShapes5 Dataset (contd.)



(a) random disks



(b) with bbox annotations

- This dataset is available through the archive `datasets_for_DLStudio.tar.gz` at the link "Download the image datasets for the main DLStudio Class" at the top of the main documentation page for DLStudio. If you install the dataset as recommended, you will see the following data archive files in the "data' subdirectory of the "Examples" directory of the DLStudio distribution:
  - 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:

```
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 image classes, cat and dog.
- The new inner class `CustomDataLoading` of the `DLStudio` platform presents a custom dataloader for the `PurdueShapes5` dataset. This dataloader understands the data format presented on Slide 55.
- 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 used for object detection and localization with the DLStudio platform.**

# The LOADnet2 Network from DLStudio

```
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, the convo layers
        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)
        )
        ##  for regression, the fc layers
        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 DLStudio's LOADnet2 Network

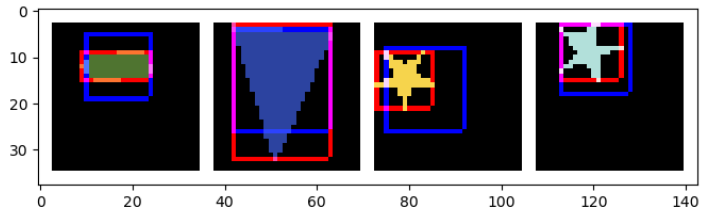
The code shown below is from DLStudio's inner class named

DetectAndLocalize.

```
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
```

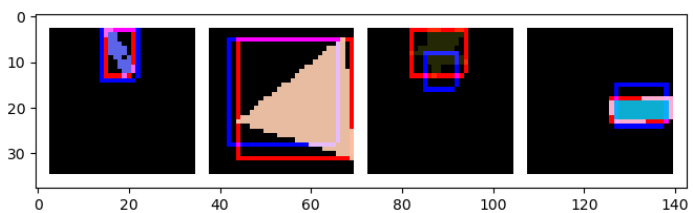
# The Two Losses vs. the Iterations During Training

```
[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
```



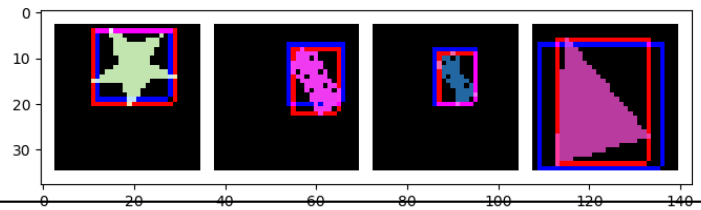

---

```
[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:  [19,12,30,21]
[epoch:1/2  iter=1000  elapsed_time=   95 secs]       loss_labelling 0.676        loss_regression: 9.106
```

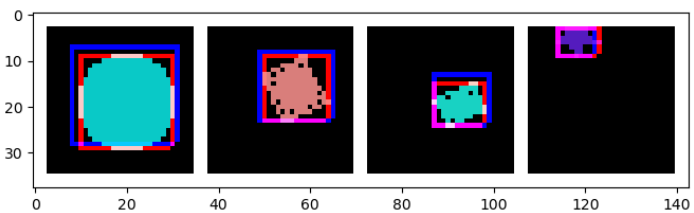


# The Two Losses vs. the Iterations During Training (contd.)

```
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=  156 secs]       loss_labelling 0.602        loss_regression: 5.423
```

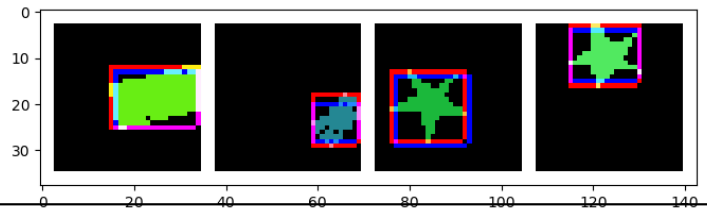



```
[epoch:1/2  iter=2000  elapsed_time=  263 secs]      Ground Truth:           disk  rectangle  rectangle       star
[epoch:1/2  iter=2000  elapsed_time=  263 secs]  Predicted Labels:           disk  rectangle       oval       star
                  gt_bb:  [7,6,27,26]
                pred_bb:  [5,4,28,25]
                  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
```

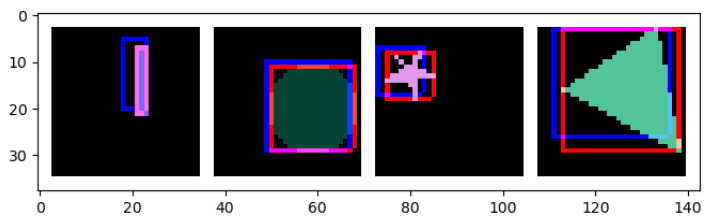


# The Two Losses vs. the Iterations During Training (contd.)

```
[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
```

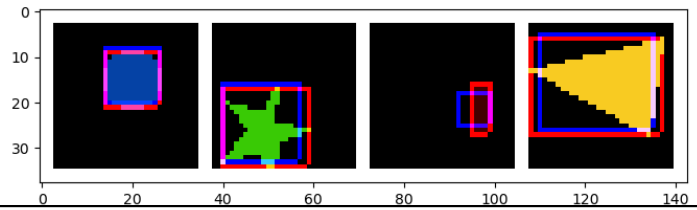



```
[epoch:2/2  iter= 500  elapsed_time=  409 secs]      Ground Truth:           oval       disk       star   triangle
[epoch:2/2  iter= 500  elapsed_time=  409 secs]  Predicted Labels:           oval       disk       star   triangle
                    gt_bb:  [18,4,20,18]
                  pred_bb:  [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
```

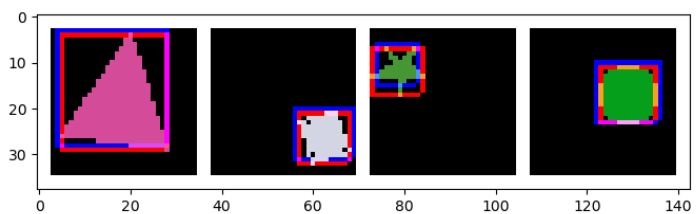


# The Two Losses vs. the Iterations During Training (contd.)

```
[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
```

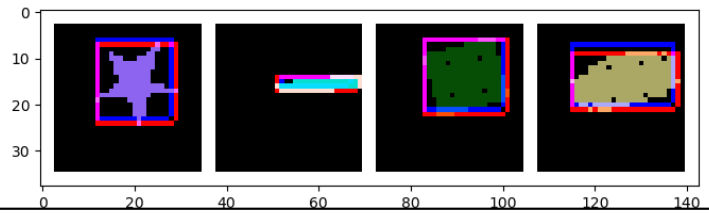



```
[epoch:2/2  iter=1500  elapsed_time=  599 secs]      Ground Truth:       triangle  rectangle       star       disk
[epoch:2/2  iter=1500  elapsed_time=  599 secs]  Predicted Labels:       triangle  rectangle       star       disk
                    gt_bb:  [2,1,25,26]
                  pred_bb:  [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
```

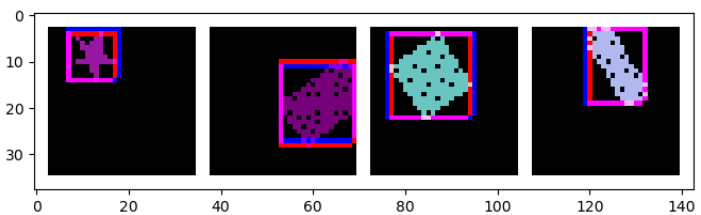


# The Two Losses vs. the Iterations During Training (contd.)

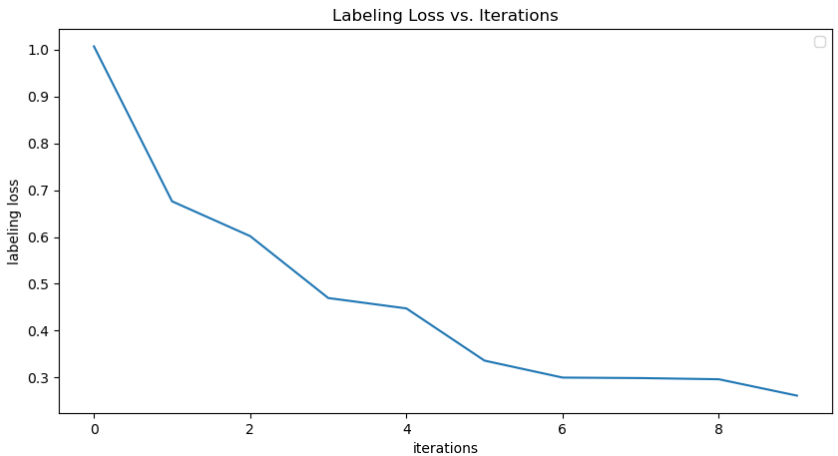
```
[epoch:2/2  iter=2000  elapsed_time=  657 secs]      Ground Truth:          star       oval  rectangle       oval
[epoch:2/2  iter=2000  elapsed_time=  657 secs]  Predicted Labels:          star       oval       disk       oval
                    gt_bb:  [9,4,26,21]
                  pred_bb:  [9,3,25,20]
                    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
```



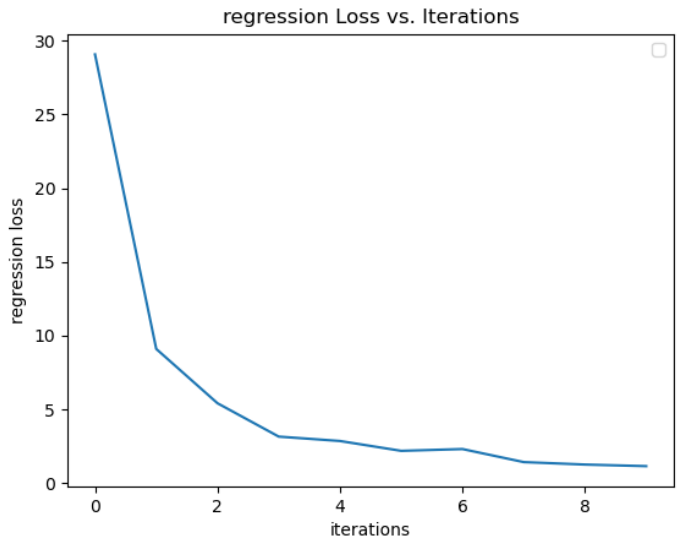

```
[epoch:2/2  iter=2500  elapsed_time=  754 secs]      Ground Truth:          star  rectangle  rectangle       oval
[epoch:2/2  iter=2500  elapsed_time=  754 secs]  Predicted Labels:          star       oval  rectangle       oval
                    gt_bb:  [4,1,14,11]
                  pred_bb:  [4,0,15,11]
                    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



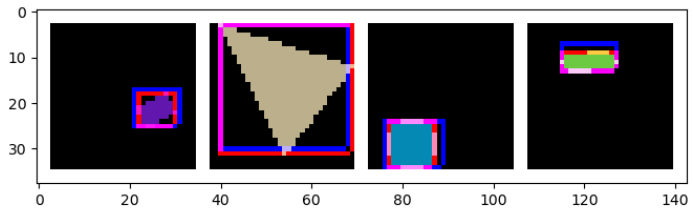

(a) labeling loss vs. iterations



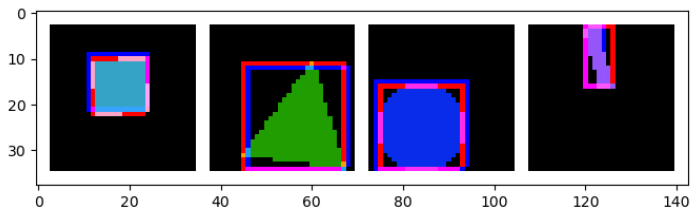

(b) regression loss vs. iterations

# 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]
```



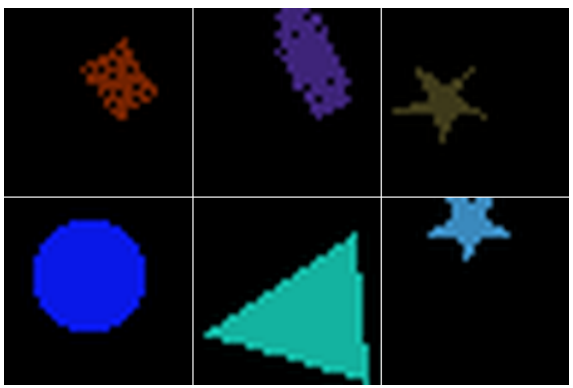
# Classification Accuracy on the Unseen Test Data (After 2 Epochs of Training)

```
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
```

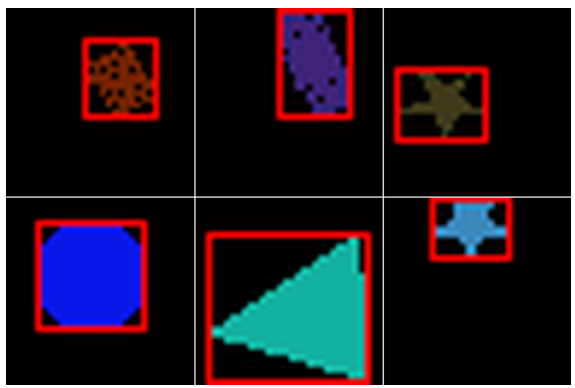
# Outline

# The Single Instance Case was Baby Steps for Object Detection

- In the first part of this lecture, the images in the dataset I used to demonstrate object detection and localization looked like those shown below.
- What makes these images almost silly is that each image contains just one "object" and there is no structured clutter in the background. Even with random noise added, these images are good only as a first exercise in object detection and localization.



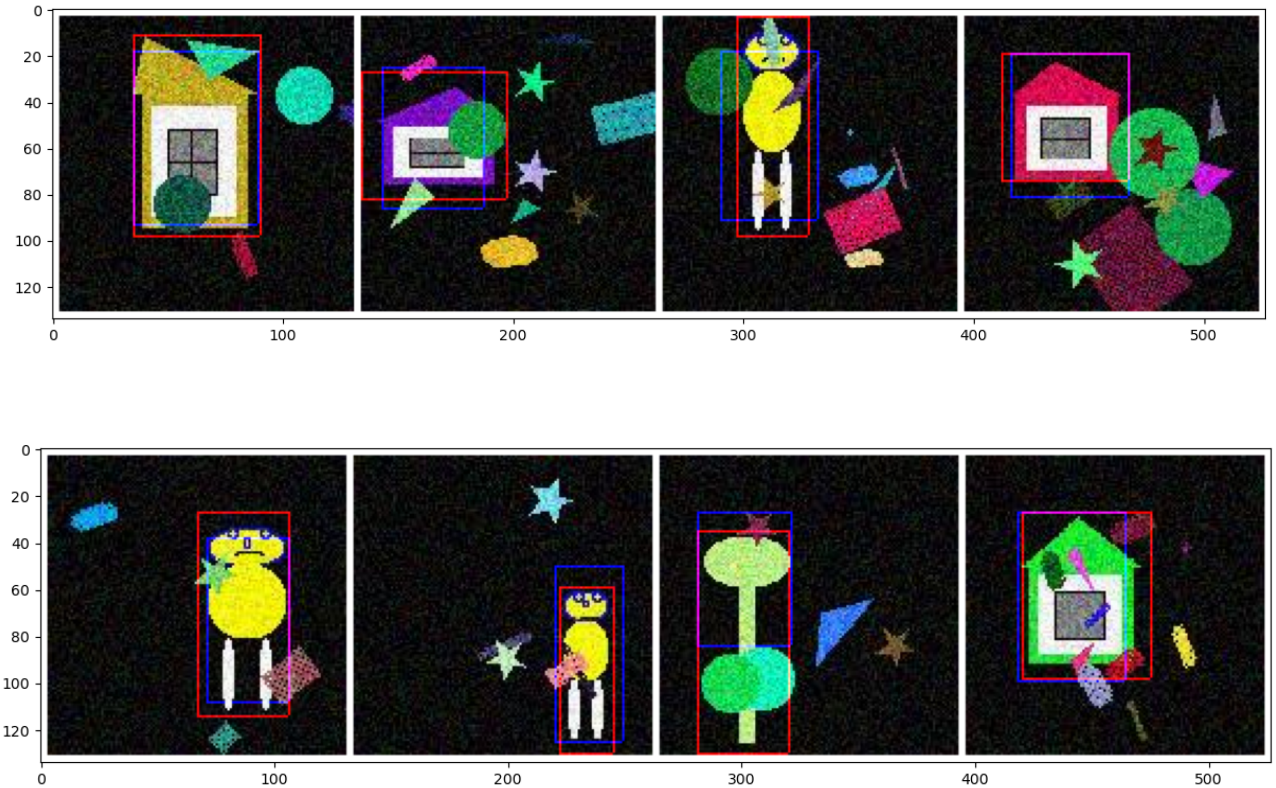
(a) objects in images



(b) with bbox annotations

# Making Object Detection a Bit More Difficult

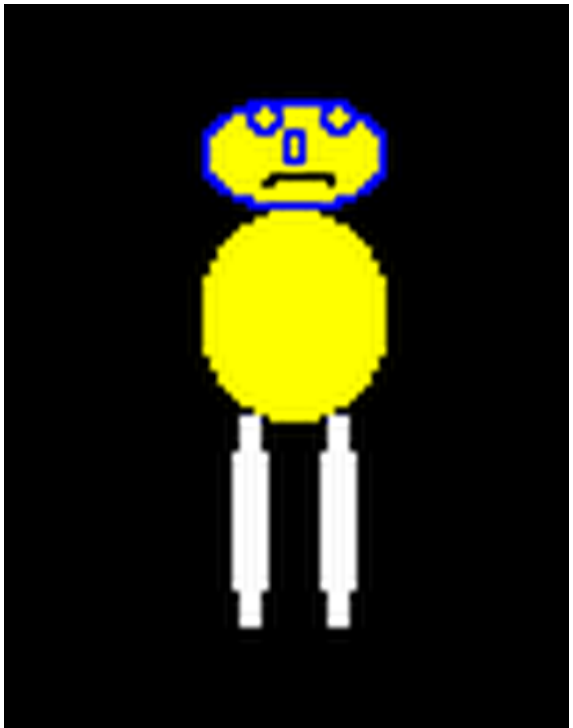
- I am now going to present a more difficult dataset of images for object detection and localization. Before I describe the objects of interest in these images, shown below are image from a couple of batches during training time:



# Introducing You to Purdue's Dr Eval Dataset

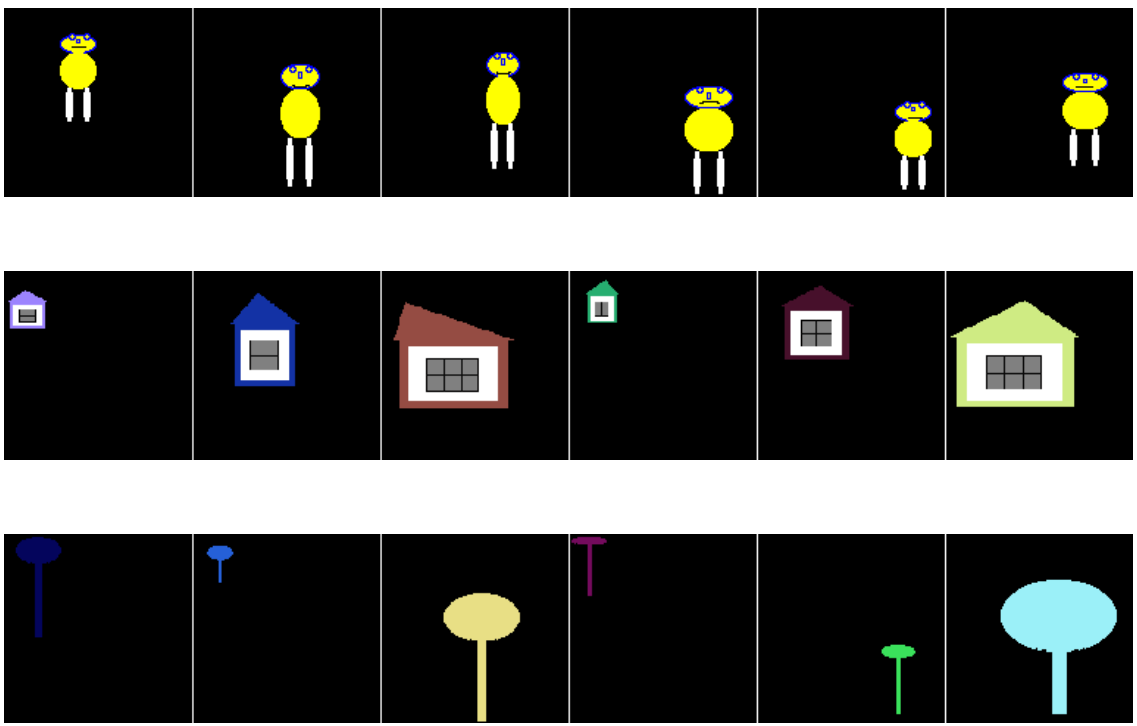
- First of all, here's the reason for the name "Dr Eval Dataset": [**In the middle of the pandemic in the winter of 2021, my wife and I decided to watch again the very old Austin Powers movies. You might be too young to realize that these movies were a great spoof of the James Bond genre of movies. The main villain in these movies is a character named "Dr. Evil" whose primary goal is to destroy the good guy Austin Powers. Around the same time I was looking for a name for a new object-detection dataset I was creating that would be more challenging than the PurdueShapes5 dataset you saw earlier.**]

- Since "Eval" is such an important word for those of us who love programming computers, I figured Dr. Eval would be an appropriate name for the dataset. Here is Dr. Eval:



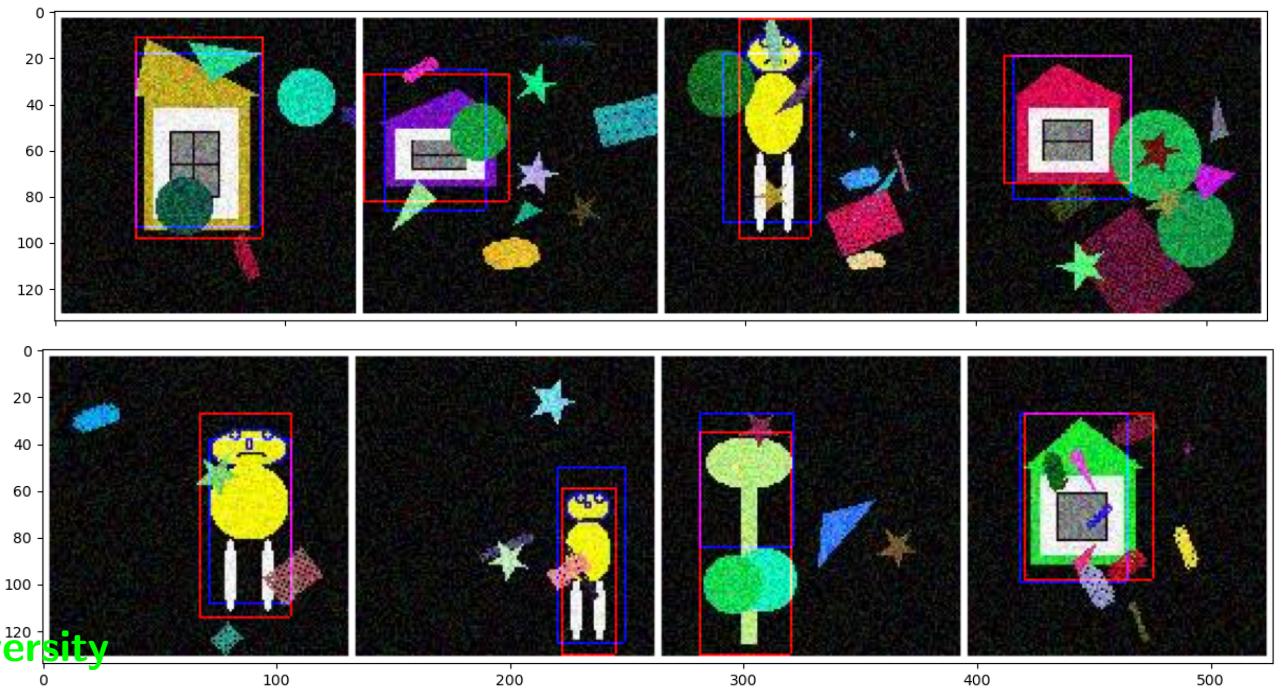
# The Dr Eval Dataset in YOLOLogic (contd.)

- Besides Dr. Eval, the dataset also contains a couple of other "objects" of interest: his house and a watertower in his neighborhood.
- All three objects are randomized with respect to their scale, aspect ratio, and other appearance variables.



# The Dr Eval Dataset in YOLOLogic (contd.)

- One thing that is common among all the objects of interest in the dataset is that they are all "oriented"— you could say they are vertically oriented. A house may not look like a house unless its base is mostly horizontal. And the same goes for the watertower.
- To the randomized object images shown on the previous slide, I add structured (but "non-oriented" artifacts) in order to produce the following sorts of images:



# The Dr Eval Dataset in YOLOLogic (contd.)

- The single-object-instance Dr Eval Dataset can be downloaded by first downloading the archive

```
datasets_for_YOLO.tar.gz
```

  through the link "Download the image datasets for YOLO" at the main webpage for the YOLOLogic module and storing the archive in the Examples directory of your install of the YOLOLogic module.

- Subsequently, execute the following command in the ExamplesObjectDetection directory:

```
tar zxvf datasets_for_YOLO.tar.gz
```

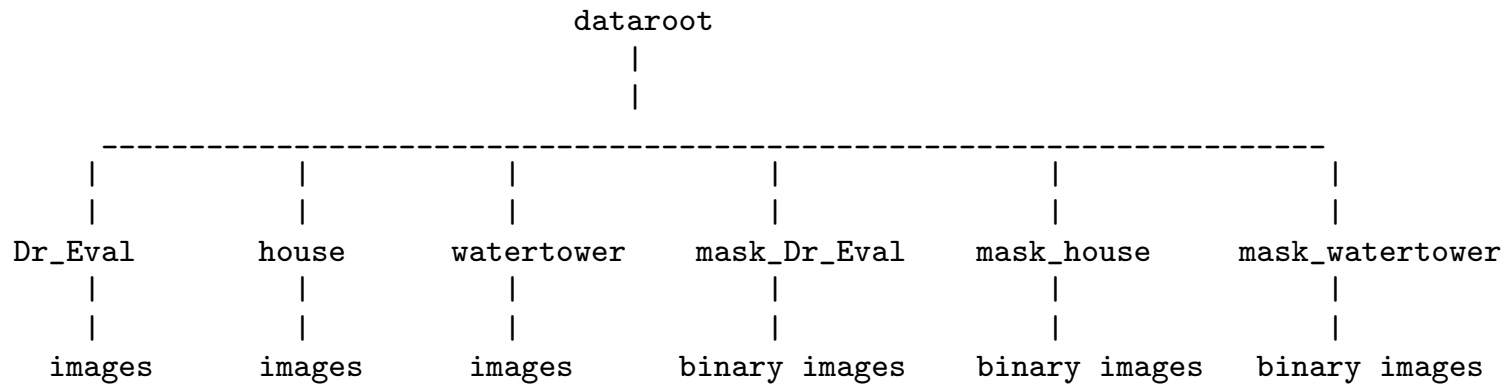
  That will create a 'data' subdirectory in the ExamplesObjectDetection directory and deposit the following datasets in it:

```
Purdue_Dr_Eval_Dataset-clutter-10-noise-20-size-10000-train.gz
Purdue_Dr_Eval_Dataset-clutter-10-noise-20-size-1000-test.gz
```

- The naming convention used for the archives: [**The string 'clutter-10' means that each image has at most 10 clutter objects in it, and the string 'noise-20' means that I have added 20% Gaussian noise to each image. The string 'size-10000' means that the dataset consists of 10,000 images.**]

# Dataloader for The Dr Eval Dataset

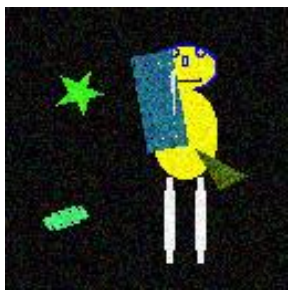
- In order to understand the implementation of the dataloader for the Dr Eval dataset for single-instance-based object detection, note that the top-level directory for the dataset is organized as follows:



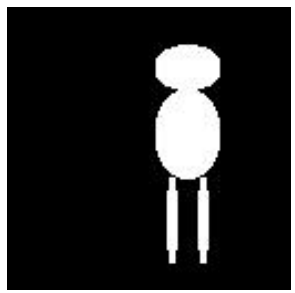

- As you can see, the three main image directories are `Dr_Eval`, `house`, and `watertower`. For each image in each of these directories, the mask for the object of interest is supplied in the corresponding directory whose name carries the prefix `mask`.

# Dataloader for The Dr Eval Dataset in YOLOLogic (contd.)

- For example, if you have an image named `29.jpg` in the `Dr_Eval` directory, you will have an image of the same name in the `mask_Dr_Eval` directory that will just be the mask for the Dr_Eval object in the former image:



(a) An example Dr_Eval image



(b) The corresponding mask image

- As you can see, the dataset does not directly provide the bounding boxes for object localization. So the implementation of the `__getitem__()` function in the dataloader must include code that calculates the bounding boxes from the masks. This you can see in the definition of the dataloader in the next three slides.

# Dataloader for The Dr Eval Dataset in YOLOLogic (contd.)

- Since this is a "non-standard" organization of the of data, the dataloader must also provide for the indexing of the images so that they can be subject to the fresh randomization that is carried out by PyTorch's `torch.utils.data.DataLoader` for each epoch of training. The code that is shown in the next three slides include the `index_dataset()` function for this purpose

- After the dataset is downloaded for the first time, the `index_dataset()` function stores away the information as a PyTorch ".pt" file so that it can be downloaded almost instantaneously at subsequent attempts.

- One final note about the dataset: Under the hood, the dataset consists of the pathnames to the image files — and NOT the images themselves. It is the job of the multi-threaded "workers" provided by `torch.utils.data.DataLoader` to actually download the images from those pathnames.

# Dataloader for The Dr Eval Dataset in YOLOLogic

```
class PurdueDrEvalDataset(torch.utils.data.Dataset):
    def __init__(self, yolo, train_or_test, dataroot_train=None, dataroot_test=None, transform=None):
        super(YOLOLogic.PurdueDrEvalDataset, self).__init__()
        self.yolo = yolo
        self.train_or_test = train_or_test
        self.dataroot_train = dataroot_train
        self.dataroot_test  = dataroot_test
        self.database_train = {}
        self.database_test = {}
        self.dataset_size_train = None
        self.dataset_size_test = None
        if train_or_test == 'train':
            self.training_dataset = self.index_dataset()
        if train_or_test == 'test':
            self.testing_dataset = self.index_dataset()
        self.class_labels = None

    def index_dataset(self):
        if self.train_or_test == 'train':
            dataroot = self.dataroot_train
        elif self.train_or_test == 'test':
            dataroot = self.dataroot_test
        entry_index = 0
        if self.train_or_test == 'train' and dataroot == self.dataroot_train:
            if '10000' in self.dataroot_train and os.path.exists("torch_saved_Purdue_Dr_Eval_dataset_train_10000.pt"):
                print("\nLoading training data from torch saved file")
                self.database_train = torch.load("torch_saved_Purdue_Dr_Eval_dataset_train_10000.pt")
                self.dataset_size_train =  len(self.database_train)
            else:
                print("""\n\n\nLooks like this is the first time you will be loading in\n"""
                      """the dataset for this script. First time loading could take\n"""
                      """up to 3 minutes.  Any subsequent attempts will only take\n"""
                      """a few seconds.\n\n\n""")
                if os.path.exists(dataroot):
                    files = glob.glob(dataroot + "/*")
                    files = [os.path.split(file)[1] for file in files]
                    class_names = sorted([file for file in files if not file.startswith("mask")])
                    if self.train_or_test == 'train':
                        self.class_labels = class_names
                    image_label_dict = {class_names[i] : i for i in range(len(class_names))}
                    for image_class in class_names:
                        image_names = glob.glob(dataroot + image_class + "/*")
                        for image_name in image_names:
                            image_real_name = os.path.split(image_name)[-1]
                            mask_name = dataroot + "mask_" + image_class + "/" + image_real_name
                            if self.train_or_test == 'train':
                                self.database_train[entry_index] = [image_label_dict[image_class], image_name, mask_name]
                            elif self.train_or_test == 'test':
                                self.database_test[entry_index] = [image_label_dict[image_class], image_name, mask_name]
                            entry_index += 1
```

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

# Dataloader for The Dr Eval Dataset

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

```
                if self.train_or_test == 'train':
                    all_training_images = list(self.database_train.values())
                    random.shuffle(all_training_images)
                    self.database_train = {i : all_training_images[i] for i in range(len(all_training_images))}
                    torch.save(self.database_train, "torch_saved_Purdue_Dr_Eval_dataset_train_10000.pt")
                    self.dataset_size_train = entry_index
                else:
                    all_testing_images = list(self.database_test.values())
                    random.shuffle(all_testing_images)
                    self.database_test = {i : all_testing_images[i] for i in range(len(all_testing_images))}
                    self.dataset_size_test = entry_index
    else:
        if os.path.exists(dataroot):
            files = glob.glob(dataroot + "/*")
            files = [os.path.split(file)[1] for file in files]
            class_names = sorted([file for file in files if not file.startswith("mask")])
            image_label_dict = {class_names[i] : i for i in range(len(class_names))}
            for image_class in class_names:
                image_names = glob.glob(dataroot + image_class + "/*")
                for image_name in image_names:
                    image_real_name = os.path.split(image_name)[-1]
                    mask_name = dataroot + "mask_" + image_class + "/" + image_real_name
                    if self.train_or_test == 'train':
                        self.database_train[entry_index] = [image_label_dict[image_class], image_name, mask_name]
                    elif self.train_or_test == 'test':
                        self.database_test[entry_index] = [image_label_dict[image_class], image_name, mask_name]
                    entry_index += 1
            if self.train_or_test == 'train':
                self.dataset_size_train = entry_index
            if self.train_or_test == 'test':
                self.dataset_size_test = entry_index
            if self.train_or_test == 'train':
                all_training_images = self.database_train.values()
                random.shuffle(all_training_images)
                self.database_train = {i : all_training_images[i] for i in range(len(all_training_images))}
                torch.save(self.database_train, "torch_saved_Purdue_Dr_Eval_dataset_train_10000.pt")
                self.dataset_size_train = entry_index
            else:
                all_testing_images = list(self.database_test.values())
                random.shuffle(all_testing_images)
                self.database_test = {i : all_testing_images[i] for i in range(len(all_testing_images))}

def __len__(self):
    if self.train_or_test == 'train':
        return self.dataset_size_train
    elif self.train_or_test == 'test':
        return self.dataset_size_test
```

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

# Dataloader for The Dr Eval Dataset

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

```
def __getitem__(self, idx):
    if self.train_or_test == 'train':
        image_label, image_name, mask_name = self.database_train[idx]
    elif self.train_or_test == 'test':
        image_label, image_name, mask_name = self.database_test[idx]
    im = Image.open(image_name)
    mask = Image.open(mask_name)
    mask_data = mask.getdata()
    non_zero_pixels = []
    for k,pixel_val in enumerate(mask_data):
        x = k % self.yolo.image_size[1]
        y = k // self.yolo.image_size[0]
        if pixel_val != 0:
            non_zero_pixels.append((x,y))
    ## x-coord increases to the left and y-coord increases going downward;  origin at upper-left
    x_min = min([pixel[0] for pixel in non_zero_pixels])
    x_max = max([pixel[0] for pixel in non_zero_pixels])
    y_min = min([pixel[1] for pixel in non_zero_pixels])
    y_max = max([pixel[1] for pixel in non_zero_pixels])
    bbox = [x_min,y_min,x_max,y_max]
    im_tensor = tvt.ToTensor()(im)
    mask_tensor = tvt.ToTensor()(mask)
    bbox_tensor = torch.tensor(bbox, dtype=torch.float)
    return im_tensor,mask_tensor,bbox_tensor,image_label
```

# The Network Class for Dr Eval Dataset in YOLOLogic

- The network class for the single-instance based detection in the Dr_Eval dataset images is an adaptation of the of the `LOADnet2` class in DLStudio. The original `LOADnet2` class was meant for the processing of $32 \times 32$ images. But now we have $128 \times 128$ images. The building-block is still the `SkipBlock` class that you saw earlier.
- The network definition has two parts: the part for classification of the image and the part for the regression needed for estimating the four numerical parameters that define the object bounding box. Just for the sake of doing so, I have used the `nn.Sequential` container for the defining the regression section. Whereas the bulk of the work in the classification section is done through the convolutional layers, the bulk of the work for regression is done by the fully-connected layers.
- If you run the script `single_instance_detection.py` in the Examples directory of the `YOLOLogic` module, you'll see that this network consists of 168 layers and 85,191,591 learnable parameters.

# The Network Class for Dr Eval Dataset in YOLOLogic (contd.)

```
class LOADnet(nn.Module):
    """
    The acronym 'LOAD' stands for 'LOcalization And Detection'.
    """
    def __init__(self, skip_connections=True, depth=8):
        super(YOLOLogic.SingleInstanceDetector.LOADnet2, self).__init__()
        if depth not in [8,10,12,14,16]:
            sys.exit("LOADnet has only been tested for 'depth' values 8, 10, 12, 14, and 16")
        self.depth = depth // 2
        self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.bn1  = nn.BatchNorm2d(64)
        self.bn2  = nn.BatchNorm2d(128)
        self.bn3  = nn.BatchNorm2d(256)
        self.skip64_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip64_arr.append(YOLOLogic.SingleInstanceDetector.SkipBlock(64, 64,skip_connections=skip_connections))
        self.skip64ds = YOLOLogic.SingleInstanceDetector.SkipBlock(64,64,downsample=True,
                                                                   skip_connections=skip_connections)
        self.skip64to128 = YOLOLogic.SingleInstanceDetector.SkipBlock(64, 128,
                                                                   skip_connections=skip_connections )
        self.skip128_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128_arr.append(YOLOLogic.SingleInstanceDetector.SkipBlock(128,128,
                                                                   skip_connections=skip_connections))
        self.skip128ds = YOLOLogic.SingleInstanceDetector.SkipBlock(128,128,
                                                 downsample=True, skip_connections=skip_connections)
        self.skip128to256 = YOLOLogic.SingleInstanceDetector.SkipBlock(128, 256,
                                                                   skip_connections=skip_connections )
        self.skip256_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip256_arr.append(YOLOLogic.SingleInstanceDetector.SkipBlock(256,256,
                                                                   skip_connections=skip_connections))
        self.skip256ds = YOLOLogic.SingleInstanceDetector.SkipBlock(256,256,
                                                   downsample=True, skip_connections=skip_connections)
        self.fc1 =  nn.Linear(8192, 1000)
        self.fc2 =  nn.Linear(1000, 3)
        ##  the rest of this __init__() is for regression
        self.conv_seqn = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2,2),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2,2)
        )
        self.fc_seqn = nn.Sequential(
            nn.Linear(65536, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, 4)
        )
```

# The Network Class for Dr Eval Dataset (contd.)

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

```
def forward(self, x):
    x = self.pool(torch.nn.functional.relu(self.conv1(x)))
    xR = x.clone()
    ## The labeling section:
    x1 = nn.MaxPool2d(2,2)(torch.nn.functional.relu(self.conv2(x)))
    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)
    x1 = x1.view(-1, 8192 )
    x1 = torch.nn.functional.relu(self.fc1(x1))
    x1 = self.fc2(x1)
    ## for bounding box regression:
    x2 = self.conv_seqn(xR)
    x2 = x2.view(x.size(0), -1)
    x2 = self.fc_seqn(x2)
    return x1,x2
```

- What follows next is the output of the function

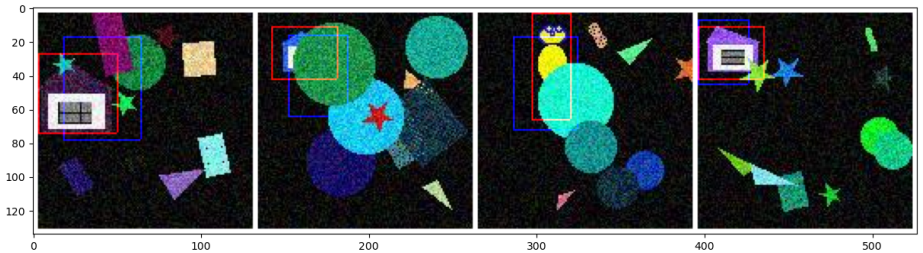
  `run_code_for_training_single_instance_detector(model, display_images=True)`

  from the class `SingleInstanceDetector` on the Dr. Eval dataset described on Slides 77 through 81 of this slide deck.

- If you don't want to see the images and witness how the predicted bounding boxes converge to the ground-truth bounding boxes, turn off the `display_images` argument shown above by setting it to `False`.

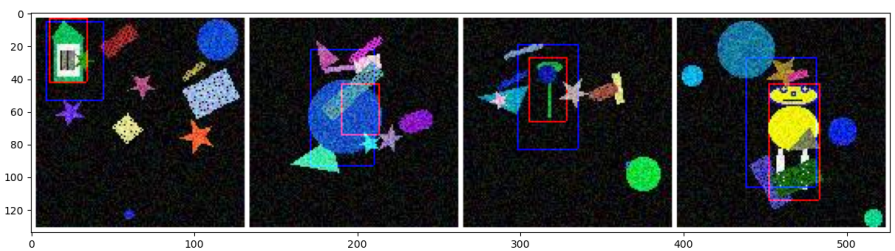
# Training the Single-Instance Detector

```
[epoch:1/6  iter= 500  elapsed_time=   61 secs]   Ground Truth:              house          house        Dr_Eval          house
[epoch:1/6  iter= 500  elapsed_time=   61 secs]   Predicted Labels:          house          house          house        Dr_Eval
                  gt_bb:  [0,24,47,71]
                pred_bb:  [15,14,61,75]
                  gt_bb:  [8,8,47,39]
                pred_bb:  [18,13,53,61]
                  gt_bb:  [32,0,55,63]
                pred_bb:  [21,14,59,69]
                  gt_bb:  [0,8,39,39]
                pred_bb:  [0,4,30,42]
[epoch:1/6  iter= 500  elapsed_time=   61 secs]   loss labeling: 1.078      loss regression: 857.606
```



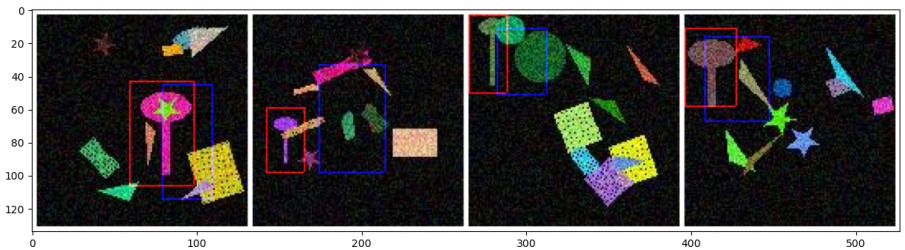

---

```
[epoch:1/6  iter=1000  elapsed_time=  155 secs]   Ground Truth:              house      watertower      watertower        Dr_Eval
[epoch:1/6  iter=1000  elapsed_time=  155 secs]   Predicted Labels:          house        Dr_Eval        Dr_Eval        Dr_Eval
                  gt_bb:  [8,0,31,39]
                pred_bb:  [6,2,41,50]
                  gt_bb:  [56,40,79,71]
                pred_bb:  [37,19,76,90]
                  gt_bb:  [40,24,63,63]
                pred_bb:  [33,16,70,80]
                  gt_bb:  [56,40,87,111]
                pred_bb:  [42,24,85,103]
[epoch:1/6  iter=1000  elapsed_time=  155 secs]   loss_labeling: 1.052      loss_regression: 328.831
```



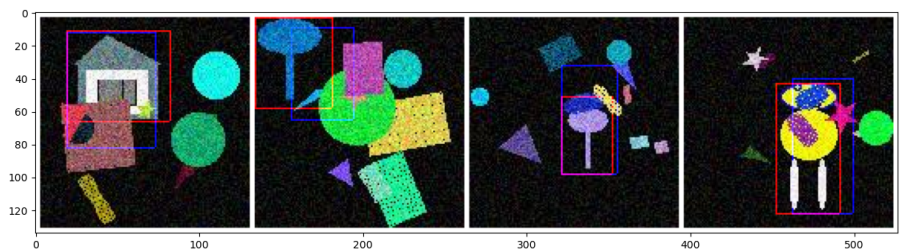
# Training the Single-Instance Detector (contd.)

```
[epoch:1/6  iter=1500  elapsed_time=  274 secs]   Ground Truth:          watertower      watertower      watertower      watertower
[epoch:1/6  iter=1500  elapsed_time=  274 secs]   Predicted Labels:        Dr_Eval         Dr_Eval         Dr_Eval      watertower
                  gt_bb:  [56,40,95,103]
                pred_bb:  [76,42,106,111]
                  gt_bb:  [8,56,31,95]
                pred_bb:  [40,30,80,95]
                  gt_bb:  [0,0,23,47]
                pred_bb:  [17,8,47,48]
                  gt_bb:  [0,8,31,55]
                pred_bb:  [12,13,51,64]
[epoch:1/6  iter=1500  elapsed_time=  274 secs]   loss_labeling: 1.023      loss_regression: 292.046
```

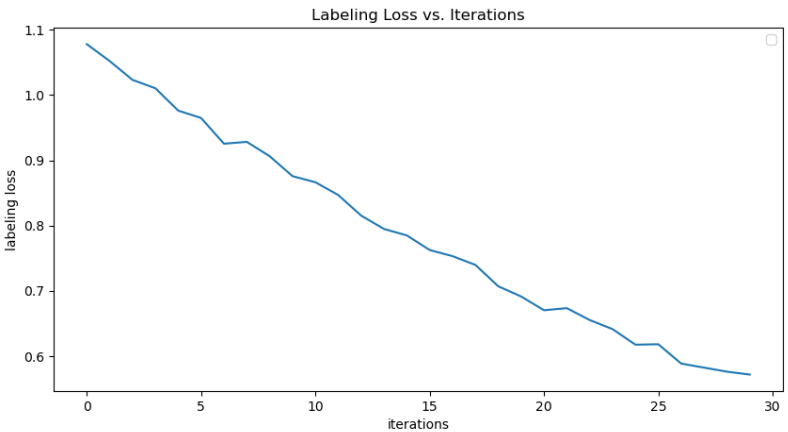



---

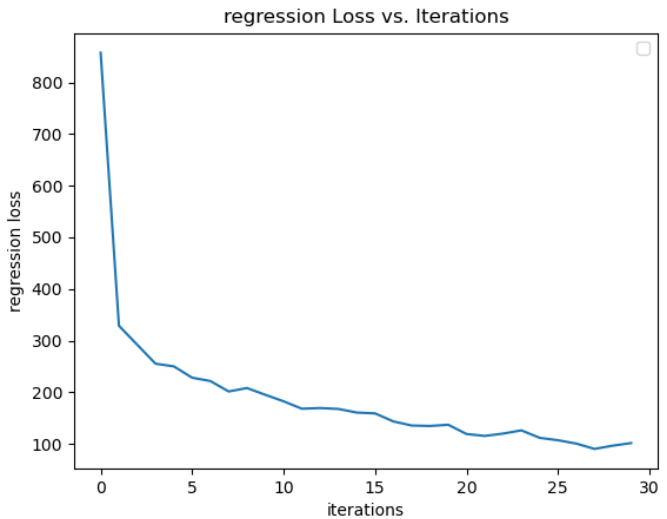
```
[epoch:1/6  iter=2000  elapsed_time=  380 secs]   Ground Truth:             house      watertower      watertower         Dr_Eval
[epoch:1/6  iter=2000  elapsed_time=  380 secs]   Predicted Labels:         house         Dr_Eval         Dr_Eval         Dr_Eval
                  gt_bb:  [16,8,79,63]
                pred_bb:  [16,9,70,79]
                  gt_bb:  [0,0,47,55]
                pred_bb:  [22,6,60,62]
                  gt_bb:  [56,48,87,95]
                pred_bb:  [56,29,90,95]
                  gt_bb:  [56,40,95,119]
                pred_bb:  [66,37,103,119]
[epoch:1/6  iter=2000  elapsed_time=  380 secs]   loss_labeling: 1.010      loss_regression: 255.298
```



# Training and Regression Losses versus Iterations




(a) labeling loss vs. iterations (6 epochs)




(b) regression loss vs. iterations (6 epochs)

# Results on Unseen Test Data

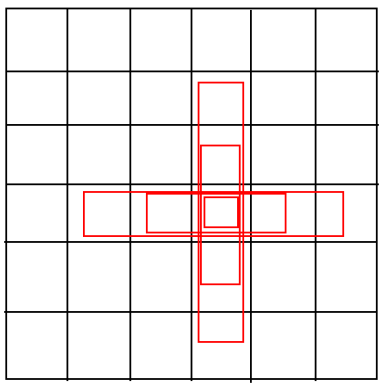
- Here is a summary of the results on the unseen test dataset after 6 epochs of training:

```
Prediction accuracy for Dr_Eval : 81 %
Prediction accuracy for house : 54 %
Prediction accuracy for watertower : 88 %


Overall accuracy of the network on the 1000 test images: 74 %


Displaying the confusion matrix:

                      Dr_Eval          house     watertower

     Dr_Eval:           81.07          10.54           8.39
       house:           38.49          54.32           7.19
  watertower:            7.90           3.77          88.33
```

# Outline

# Transitioning to Multi-Instance Object Detection

- As I mentioned in the Preamble, for multi-instance object detection, my goal in this lecture is to focus on the YOLO framework.
- Dividing an image into a grid of cells and defining a set of anchorboxes for each cell is basic to the YOLO logic. Let $S \times S$ array represent the cells in the grid thus created. The example shown below depicts a $6 \times 6$ grid that you can pretend is overlaid on an image.
- What is shown in red are the five anchorboxes for the cell at index $(3, 3)$ in the grid.



# The YOLO Logic

- As you can tell from the red anchorboxes shown on the previous slide, they are characterized by the aspect ratio – meaning the ratio of the height to the width. My implementation of the YOLO logic in `YOLOLogic` uses five anchorboxes for each cell of the $S \times S$ grid with the following aspect ratios: 1/5, 1/3, 1/1, 3/1, and 5/1.
- The job of predicting the bounding-box and the class-label for an object in the image is the responsibility of the grid cell that has the center of the object bounding-box in it. Even more specifically, in that grid cell, making these predictions is the responsibility of the anchor-box to which the object instance is assigned.
- In my implementation of the YOLO logic in `YOLOLogic`, I have defined a convenience variable `yolo_interval` that makes it easy to write several kinds of programming statements related to the creation of the grid, assignment of the true bounding boxes in the training data to the cells and the anchorboxes associated with the cells. The value of `yolo_interval` is the height (or the width) of each cell in the $S \times S$ grid.

# The YOLO Logic (contd.)

- In the implementation shown in `YOLOLogic`, I have used `yolo_interval=20` for the $128 \times 128$ images in the `PurdueDrEvalMultiDataset`.
- With `yolo_interval=20`, we end up with a $6 \times 6$ array of cells in the grid. At the moment, I have not bothered with the bottom 8 rows and the right-most 8 columns of the image that get left out of the area covered by the grid.
- In keeping with what I mentioned earlier, an object instance in a training image is assigned to that cell whose center is closest to the center of the bounding box for the instance.
- *After the cell assignment*, the instance is assigned to that anchorbox whose aspect ratio comes closest to matching the aspect ratio of the instance.

# The YOLO Logic (contd.)

- The assigning of an object instance to a $(cell, anchor\_box)$ pair is encoded in the form of a $5 + C$ element long vector where $C$ is the number of classes for the object instances. I'll refer to this $5 + C$-sized vector as the `yolo_vector`. [The next slide talks about what these $5+C$ elements stand for.]

- As you will see later, I will also be using the term `yolo_tensor` to represent all of the `yolo_vector`'s that can be created in an image through all the cells and all the anchorboxes. For obvious reasons, only a very small number of the `yolo_vector`'s in a `yolo_tensor` would contain non-zero data — depending on how many object instances exist in an image.

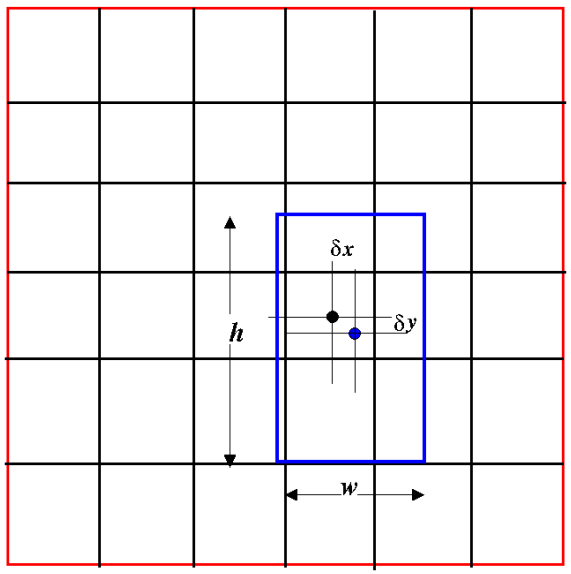
- The number $C$ of classes in the dataset `PurdueDrEvalMultiDataset` is 3. These are the `Dr_Eval` class with the class-label 0, the `house` class with the class-label 1, and the `watertower` class with the class-label 2.

# The YOLO Logic (contd.)

- With $C = 3$, in my implementation each `yolo_vector` is an 8-element vector whose last 3 elements represent the one-hot encoding for the class label associated with the object instance assigned to a given $(cell, anchor\_box)$ pair.
- As for the first five elements of the vector encoding for an anchorbox, these are set as described in what follows:
- The first element is set to 1 if an object instance in a training image was actually assigned to that anchorbox.
- The next two elements of the 8-element `yolo_vector` are the $(x, y)$ displacements of the center of the actual bounding box for the instance vis-a-vis the center of the cell.
- The two displacements mentioned above are expressed as a fraction of the width and the height of the cell as explained on the next slide.

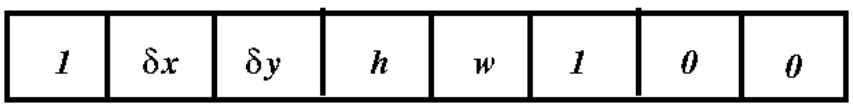
# The YOLO Logic (contd.)

- The figure shown below illustrates the $(\partial x, \partial y)$ displacements between the center of the cell and the center of the bounding box in a training image for a given object instance. The bounding box is shown with a blue outline. The displacements are between the center of the cell and the center of the bounding box.



# The YOLO Logic (contd.)

- The remaining two elements of the first five elements of the 8-element `yolo_vector` encoding are the actual height $h$ and the actual width $w$ of the true bounding box for the instance in question as a multiple of the cell dimension.
- Let's now see what goes into the last 3 elements of an 8-element `yolo_vector`: Consider the case when an instance of Dr_Eval is assigned to an anchorbox for one of the cells. As you already know, the class index for the Dr_Eval label is 0. So a 3-element one-hot representation for this class is $[1, 0, 0]$. Therefore, amongst the last 3 elements reserved for the class label, the first of the three will be set to 1. The 8-element `yolo_vector` for this anchorbox is shown below. Its first element is 1 because a true bounding box was assigned to it.



# Stuffing the Yolo Vectors in an Image into a Yolo Tensor

- The following statements reproduced from my YOLO implementation in the YOLOLogic module should give you a better understanding of the relationship between a `yolo_vector` and a `yolo_tensor`:

```
yolo_tensor = torch.zeros( batch_size, num_yolo_cells, num_anchor_boxes, 8 )
##  calculate del_x, del_y, bh, bw
yolo_vector  = torch.FloatTensor( [1.0, del_x.item(), del_y.item(), bh.item(), bw.item(), 0,0,0] )
yolo_vector[5 + class_label_of_object] = 1              ## creates a one-hot vector representation
yolo_cell_index =  cell_row_indx.item() * num_cells_image_width  +  cell_col_indx.item()
yolo_tensor[0, yolo_cell_index, anch_box_index] = yolo_vector
```

- While the logic presented above for creating a `yolo_tensor` from all the `yolo_vector`'s present in an image is correct, it needs a modification before it can be fed into a neural network: You need to add one more element to each 8-element yolo vector for the proper functioning of the cross-entropy loss. You see, we want the neural network to put its probability mass in that element (of the last 3 elements) that corresponds to the true label of the object represented by a yolo vector. But what if the cell/anchorbox combo does NOT contain any objects? Where should the probability mass go in such cases? The extra element comes in handy for this purpose.

# Stuffing the Yolo Vectors into a Yolo Tensor (contd.)

- Therefore, what follows is a more correct definition of the yolo tensor we need for each image. To make a distinction with the yolo tensor defined earlier, I call this an "augmented yolo tensor" `yolo_tensor_aug`:

```
yolo_tensor_aug = torch.zeros( batch_size, num_yolo_cells, num_anchor_boxes, 9 )
```

- When constructing the yolo tensor for a batch of training images, we must initialize the new element (the $9^{th}$ element) to a probability mass 1 when the first element of the yolo vector is set to 0:

```
for ibx in range(im_tensor.shape[0]):
    for icx in range(num_yolo_cells):
        for iax in range(num_anchor_boxes):
            if yolo_tensor_aug[ibx, icx, iax, 0] == 0:
                yolo_tensor_aug[ibx, icx, iax,-1] = 1
```

- The training images are fied into the `NetForYolo` network and predictions made by the network at its output reshaped into a tensor of the same shape as the `yolo_tensor_aug` shown above:

```
output = net(im_tensor)
predictions = output.view(batch_size, num_yolo_cells, num_anchor_boxes, 8)
```

# Comparing Ground-Truth with Predictions (contd.)

- At each iteration of the training cycle, I must compare the ground-truth `yolo_tensor` for the training image with the predicted `yolo_tensor` in the `predictions`.
- This comparison is carried out separately for every corresponding pair of `yolo_vector`'s in the ground-truth and the predicted tensors, as shown in the code on the next slide. The goal of the comparisons mentioned above is to estimate a value for the loss.
- And, when comparing the corresponding yolo vectors in the predictions vis-a-vis their targets, we must pay respect to the different semantics for the different portions of a `yolo_vector`: I use the Binary Cross-Entropy Loss (`nn.BCELoss`) for the first element of the `yolo_vector`, mean-squared-error loss (`nn.MSELoss`) for the next four numerical elements, and, finally, the regular cross-entropy loss (`nn.CrossEntropyLoss`) for the remaining elements.

# Estimating the Loss

- The code shown on Slide 107 explains the basic idea of how we want to estimate the loss. This sort of code would come into play after the following statements:

```
output = net(im_tensor)
predictions_aug = output.view(self.yolo.batch_size,num_yolo_cells,num_anchor_boxes,9)
loss = torch.tensor(0.0, requires_grad=True).float().to(self.yolo.device)
```

  where the first line is simply feeding a batch of input images into the network, the next line is reshaping the 1620-elements (36 cells $\times$ 5 anchorboxes $\times$ 9-element Yolo vector $=$ 1620) of the output of the network into the same shape as the yolo_tensor representation of the input image, and the third line is for initializing the loss tensor.

- In the code shown on Slide 107, lines (65) through (83) visit every corresponding pair of `yolo_vector`'s in the ground-truth and the predicted `yolo_tensor`'s for each training image in a batch.

# Estimating the Loss (contd.)

- Continuing with the explanation of the loss calculation shown on the next slide, lines (69) through (72) estimate the `nn.BCELoss` (Binary Cross-Entropy Loss) for the first element values in the corresponding pair of `yolo_vector`'s.
- Lines (73) through (77) apply the `nn.MSELoss` to the regression parameters in the next four elements of the two `yolo_vector`'s.
- Lines (78) through (82) apply the `nn.CrossEntropyLoss` to the remaining elements for estimating the labeling loss between the two `yolo_vector`'s.
- **IMPORTANT:** The code shown on the next slide for calculating the loss is as it existed in versions older than 2.1.0 of YOLOLogic. In Version 2.1.0, I got rid of the "for-loops" and that code is on Slide 108. The older code on the next slide is still useful for understanding the basic logic of how the loss is calculated for each cell and each anchor box.

# Estimating the Loss (contd.)

```
criterion1 = nn.BCELoss()                                                                    ## (3)
criterion2 = nn.MSELoss()                                                                    ## (4)
criterion3 = nn.CrossEntropyLoss()                                                           ## (5)
...
...
output = net(im_tensor)                                                                      ## (62)
predictions = output.view( .... shaped the same as the yolo_tensor for the training image ....)   ## (63)
loss = torch.tensor(0.0, requires_grad=True).float().to(self.yolo.device)                    ## (64)
for icx in range(num_yolo_cells):                                                            ## (65)
    for iax in range(num_anchor_boxes):                                                      ## (66)
        pred_yolo_vector = predictions_aug[0,icx,iax]                                        ## (67)
        target_yolo_vector = yolo_tensor_aug[0,icx,iax]                                      ## (68)
        ##  Estimating presence/absence of object and the Binary Cross Entropy section:
        object_presence = nn.Sigmoid()(torch.unsqueeze(pred_yolo_vector[0], dim=0))          ## (69)
        target_for_prediction = torch.unsqueeze(target_yolo_vector[0], dim=0)                ## (70)
        bceloss = criterion1(object_presence, target_for_prediction)                        ## (71)
        loss  +=  bceloss                                                                    ## (72)
        ## MSE section for regression params:
        pred_regression_vec = pred_yolo_vector[1:5]                                          ## (73)
        pred_regression_vec = torch.unsqueeze(pred_regression_vec, dim=0)                    ## (74)
        target_regression_vec = torch.unsqueeze(target_yolo_vector[1:5], dim=0)              ## (75)
        regression_loss = criterion2(pred_regression_vec, target_regression_vec)             ## (76)
        loss  += regression_loss                                                             ## (77)
        ##  CrossEntropy section for object class label:
        probs_vector = pred_yolo_vector[5:]                                                  ## (78)
        probs_vector = torch.unsqueeze( probs_vector, dim=0 )                                ## (79)
        target = torch.argmax(target_yolo_vector[5:])                                        ## (80)
        target = torch.unsqueeze( target, dim=0 )                                            ## (81)
        class_labeling_loss = criterion3(probs_vector, target)                               ## (82)
        loss +=  class_labeling_loss                                                         ## (83)
```

# Estimating the Loss without the "for" Loops

- Now that you understand the basics of how to compute the loss for a YOLO network, it's time to see how you would actually implement it — without the "for"-loops on the previous slide. Here is the implementation in the YOLOLogic module (starting with version 2.1.0):

```
criterion1 = nn.BCELoss(reduction='sum')                                          ## (1)
criterion2 = nn.MSELoss(reduction='sum')                                          ## (2)
criterion3 = nn.CrossEntropyLoss(reduction='sum')                                 ## (3)
...
...

loss = torch.tensor(0.0, requires_grad=True).float().to(self.yolo.device)          ## (68)
##  Estimating presence/absence of object with the Binary Cross Entropy loss:
bceloss = criterion1( nn.Sigmoid()(predictions_aug[:,:,:,0]), yolo_tensor_aug[:,:,:,0] )   ## (69)
loss += bceloss                                                                   ## (70)
## MSE loss for the regression params for the bounding boxes:
regression_loss = criterion2(predictions_aug[:,:,:,1:5], yolo_tensor_aug[:,:,:,1:5])       ## (71)
loss += regression_loss                                                           ## (72)
##  CrossEntropy loss for object class labels:
targets = yolo_tensor_aug[:,:,:,5:]                                               ## (73)
targets = targets.view(-1,4)                                                      ## (74)
targets = torch.argmax(targets, dim=1)                                            ## (75)
probs = predictions_aug[:,:,:,5:]                                                 ## (76)
probs = probs.view(-1,4)                                                          ## (77)
class_labeling_loss = criterion3( probs, targets)                                 ## (78)
loss += class_labeling_loss                                                       ## (79)
```

# Estimating the Loss without the "for" Loops (contd.)

- Did you notice the option "`reduction='sum'`" in all three constructor calls in Lines (1), (2), and (3) on the previous slide?

- **The default for calculating all losses is the `reduction='mean'` option.**

  [Whether or not to use the default option depends on the constraints on the argument tensor shapes that you must observe when calling a loss function. Consider the case of the CrossEntropyLoss shown in Lines 73 through 78 on the previous slide. For the two args that this function requires: The first arg tensor must have two axes, one for the batch and the second for the output of the classifier network, which is internally subject to LogSoftmax activation (the number of elements along this axis is equal to the number of classes). I'll refer to the values in the first arg tensor as the pre-prob values. The second arg tensor must also have two axes, one for the batch and the other for the ground-truth class label — these would be the target labels. The second tensor must contain a single element for the target in each batch instance.]

- The tensor shape constraint mentioned above is the reason for reshaping the tensors in Lines (77) and (74) in the code on the previous slide. Now read the following note for why we need the `reduction='sum'` option:

  [For case of CrossEntropy loss, the default reduction means taking the mean over all results along the batch axis — the Axis 0 of what is fed into the loss calculation function. Since we have 180 cell/anchorbox combos in each image and if we assume the batch size to be 4, that would mean that the dimensionality along the batch axis after reshaping in Lines (77) and (74) would be 720. With default reduction that would make the cross-entropy loss insignificant compared to the other two losses. We are not carrying out tensor reshaping for the other two losses and, therefore, the default reduction for those two cases would not have the same consequences. We fix this problem by using the "reduction='sum'" option when we call the constructor for nn.CrossEntropyLoss. I have used this option for all three loss related constructors to put them on an equal footing.]

# Outline

# The Dr Eval Multi Dataset

- The dataset to use for experimenting with multi-instance detection is named

  `PurdueDrEvalMultiDataset`

  Note the string "Multi" in the name of the dataset. The dataset I described earlier for single-instance detection was named `PurdueDrEvalDataset`.

- The data archives that you would need to download from the webpage for the YOLOLogic module are as follows, with the first for training and the second for testing: [See Slide 56 for the instructions.]

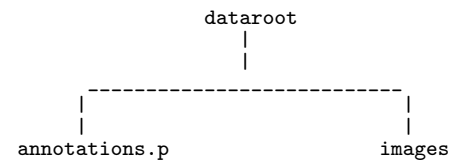
  `Purdue_Dr_Eval_Multi_Dataset-clutter-10-noise-20-size-10000-train.gz`

  `Purdue_Dr_Eval_Multi_Dataset-clutter-10-noise-20-size-1000-test.gz`

- In the naming convention for the archives, the substring "clutter-10" means that the dataset images are allowed to contain up to 10 clutter objects. And the substring "noise-20" means that 20% Gaussian noise was added to the images.

# The Dr Eval Multi Dataset (contd.)

- Continuing from the previous slide, as with the previous dataset for single-instance detection, the "multi" dataset also contains three kinds of objects in its images: "Dr_Eval", "house" and "watertower".
- Each 128x128 image contains up to 5 instances of the three types of objects. The instances are randomly scaled and colored and the number of instances chosen for each image is also random.
- Given the richness of image annotations, the dataset is organized differently from what you saw for the single-instance case. The directory structure for the dataset is as follows, with all the images in a single subdirectory:




- As for the annotations, the annotation for each $128 \times 128$ image is a dictionary that contains information related to all the object instances in the image.

# The Dr Eval Multi Dataset (contd.)

- Here is an example of the annotation for an image that has three instances in it.

```
annotation:  {'filename': None,
              'num_objects': 3,
              'bboxes': {0: (67, 72, 83, 118),
                         1: (65, 2, 93, 26),
                         2: (16, 68, 53, 122),
                         3: None,
                         4: None},
              'bbox_labels': {0: 'Dr_Eval',
                              1: 'house',
                              2: 'watertower',
                              3: None,
                              4: None},
              'seg_masks': {0: <PIL.Image.Image image mode=1 size=128x128 at 0x7F5A06C838E0>
                            1: <PIL.Image.Image image mode=1 size=128x128 at 0x7F5A06C837F0>
                            2: <PIL.Image.Image image mode=1 size=128x128 at 0x7F5A06C838B0>
                            3: None,
                            4: None}
             }
```

- The annotations for the individual images are stored in a global Python dictionary called `all_annotations` whose keys consist of the pathnames to the individual image files and the values the annotations dict for the corresponding images.

# The Dr Eval Multi Dataset (contd.)

- The annotation archive name `annotations.p` shown in the keystroke diagram on Slide 112 is what you get by calling `pickle.dump()` on the `all_annotations` dictionary.

- Shown below is an image from the dataset along with its annotation dictionary. The mask image you see on the right is a composite of four separate masks.

```
annotation:  { 'filename': None,
               'num_objects': 4,
               'bboxes': {0: (85, 9, 97, 52), 1: (0, 2, 11, 47), 2: (72, 45, 88, 93), 3: (64, 65, 71, 77), 4: None},
               'bbox_labels': {0: 'Dr_Eval', 1: 'Dr_Eval', 2: 'Dr_Eval', 3: 'watertower', 4: None},
               'seg_masks': {0: <PIL.Image.Image image mode=1 size=128x128 at 0x7FB520D42520>,
                             1: <PIL.Image.Image image mode=1 size=128x128 at 0x7FB520D429D0>,
                             2: <PIL.Image.Image image mode=1 size=128x128 at 0x7FB520D42AC0>,
                             3: <PIL.Image.Image image mode=1 size=128x128 at 0x7FB520D428B0>,
                             4: None}}
```

# Dataloader for PurdueDrEvalMultiDataset

- As shown on the next couple of slides, the implementation of a dataloader for this dataset is made easy by the fact that all images reside in a single directory as mentioned earlier on Slide 112. For the indexing part of the dataloader, all that needs to be done is to scan the image directory and to store the path to each image in a dictionary.

- Obviously, an important issue for this dataloader is extracting the annotations from the pickled archive and associating each image path with its corresponding annotation. The code for that is in the `__getitem__()` method of the dataset class defined in the next two slides.

# Dataloader for PurdueDrEvalMultiDataset (contd.)

```
class PurdueDrEvalMultiDataset(torch.utils.data.Dataset):
    def __init__(self, yolo, train_or_test, dataroot_train=None, dataroot_test=None, transform=None):
        super(YOLOLogic.PurdueDrEvalMultiDataset, self).__init__()
        self.yolo = yolo
        self.train_or_test = train_or_test
        self.dataroot_train = dataroot_train
        self.dataroot_test  = dataroot_test
        self.database_train = {}
        self.database_test = {}
        self.dataset_size_train = None
        self.dataset_size_test = None
        if train_or_test == 'train':
            self.training_dataset = self.index_dataset()
        if train_or_test == 'test':
            self.testing_dataset = self.index_dataset()
        self.class_labels = None

    def index_dataset(self):
        if self.train_or_test == 'train':
            dataroot = self.dataroot_train
        elif self.train_or_test == 'test':
            dataroot = self.dataroot_test
        if self.train_or_test == 'train' and dataroot == self.dataroot_train:
            if '10000' in self.dataroot_train and os.path.exists("torch_saved_Purdue_Dr_Eval_multi_dataset_train_10000.pt"):
                print("\nLoading training data from torch saved file")
                self.database_train = torch.load("torch_saved_Purdue_Dr_Eval_multi_dataset_train_10000.pt")
                self.dataset_size_train =  len(self.database_train)
            else:
                print("""\n\n\nLooks like this is the first time you will be loading in\n"""
                      """the dataset for this script. First time loading could take\n"""
                      """up to 3 minutes.  Any subsequent attempts will only take\n"""
                      """a few seconds.\n\n\n""")
                if os.path.exists(dataroot):
                    all_annotations = pickle.load( open( dataroot + '/annotations.p', 'rb') )
                    all_image_paths = sorted(glob.glob(dataroot + "images/*"))
                    all_image_names = [os.path.split(filename)[1] for filename in all_image_paths]
                    for idx,image_name in enumerate(all_image_names):
                        annotation = all_annotations[image_name]
                        image_path = dataroot + "images/" + image_name
                        self.database_train[idx] = [image_path, annotation]
                    all_training_images = list(self.database_train.values())
                    random.shuffle(all_training_images)
                    self.database_train = {i : all_training_images[i] for i in range(len(all_training_images))}
                    torch.save(self.database_train, "torch_saved_Purdue_Dr_Eval_multi_dataset_train_10000.pt")
                    self.dataset_size_train = len(all_training_images)
```

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

# Dataloader for PurdueDrEvalMultiDataset (contd.)

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

```
        elif self.train_or_test == 'test' and dataroot == self.dataroot_test:
            if os.path.exists(dataroot):
                all_annotations = pickle.load( open( dataroot + '/annotations.p', 'rb') )
                all_image_paths = sorted(glob.glob(dataroot + "images/*"))
                all_image_names = [os.path.split(filename)[1] for filename in all_image_paths]
                for idx,image_name in enumerate(all_image_names):
                    annotation = all_annotations[image_name]
                    image_path = dataroot + "images/" + image_name
                    self.database_test[idx] =  [image_path, annotation]
                all_testing_images = list(self.database_test.values())
                random.shuffle(all_testing_images)
                self.database_test = {i : all_testing_images[i] for i in range(len(all_testing_images))}
                self.dataset_size_test = len(all_testing_images)

    def __len__(self):
        if self.train_or_test == 'train':
            return self.dataset_size_train
        elif self.train_or_test == 'test':
            return self.dataset_size_test

    def __getitem__(self, idx):
        if self.train_or_test == 'train':
            image_path, annotation = self.database_train[idx]
        elif self.train_or_test == 'test':
            image_path, annotation = self.database_test[idx]
        im = Image.open(image_path)
        im_tensor = tvt.ToTensor()(im)
        seg_mask_tensor = torch.zeros(5,128,128)
        bbox_tensor     = torch.zeros(5,4, dtype=torch.uint8)
        bbox_label_tensor    = torch.zeros(5, dtype=torch.uint8) + 13
        num_objects_in_image = annotation['num_objects']
        obj_class_labels = sorted(self.yolo.class_labels)
        self.obj_class_label_dict = {obj_class_labels[i] : i for i in range(len(obj_class_labels))}
        for i in range(num_objects_in_image):
            bbox = annotation['bboxes'][i]
            seg_mask = annotation['seg_masks'][i]
            bbox     = annotation['bboxes'][i]
            label    = annotation['bbox_labels'][i]
            bbox_label_tensor[i] = self.obj_class_label_dict[label]
            seg_mask_arr = np.array(seg_mask)
            seg_mask_tensor[i] = torch.from_numpy(seg_mask_arr)
            bbox_tensor[i] = torch.LongTensor(bbox)
        return im_tensor, seg_mask_tensor, bbox_tensor, bbox_label_tensor, num_objects_in_image
```

# Outline

# NetForYolo — A Network for Multi-Instance Object Detection

- The multi-instance object detection network shown on the next slide predicts what would be a flattened representation of the `yolo_tensor` for a training images.
- A $6 \times 6$ gridding of the images, using 5 anchorboxes in each cell of the grid, and 9-elements for the augmented `yolo_vector` encodings results in a 1620-element flattened representation of the `yolo_tensor_aug` for a training image. The job of the network shown on the next slide is to predict this 1620-element vector for each training image. As you know by this time, for loss calculations, we reshape this flat vector into the same shape as the ground-truth `yolo_tensor_aug` for a training image.
- The final convolutional layer in the network shown on the next slide produces an output with 8192 nodes. Subsequently, a fully-connected section of the network maps those to 1620 nodes for the required regression.

# NetForYolo (contd.)

```
class NetForYolo(nn.Module):
    def __init__(self, skip_connections=True, depth=8):
        super(YOLOLogic.YoloLikeDetector.NetForYolo, self).__init__()
        if depth not in [8,10,12,14,16]:
            sys.exit("This network has only been tested for 'depth' values 8, 10, 12, 14, and 16")
        self.depth = depth // 2
        self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.bn1  = nn.BatchNorm2d(64)
        self.bn2  = nn.BatchNorm2d(128)
        self.bn3  = nn.BatchNorm2d(256)
        self.skip64_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip64_arr.append(YOLOLogic.YoloLikeDetector.SkipBlock(64, 64, skip_connections=skip_connections))
        self.skip64ds = YOLOLogic.YoloLikeDetector.SkipBlock(64,64,downsample=True, skip_connections=skip_connections)
        self.skip64to128 = YOLOLogic.YoloLikeDetector.SkipBlock(64, 128, skip_connections=skip_connections )
        self.skip128_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128_arr.append(YOLOLogic.YoloLikeDetector.SkipBlock(128,128,skip_connections=skip_connections))
        self.skip128ds = YOLOLogic.YoloLikeDetector.SkipBlock(128,128,downsample=True, skip_connections=skip_connections)
        self.skip128to256 = YOLOLogic.YoloLikeDetector.SkipBlock(128, 256, skip_connections=skip_connections )
        self.skip256_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip256_arr.append(YOLOLogic.YoloLikeDetector.SkipBlock(256,256,skip_connections=skip_connections))
        self.skip256ds = YOLOLogic.YoloLikeDetector.SkipBlock(256,256,downsample=True, skip_connections=skip_connections)
        self.fc_seqn = nn.Sequential(
            nn.Linear(8192, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 2048),
            nn.ReLU(inplace=True),
            nn.Linear(2048, 1620)
        )

    def forward(self, x):
        x = self.pool(torch.nn.functional.relu(self.conv1(x)))
        x = nn.MaxPool2d(2,2)(torch.nn.functional.relu(self.conv2(x)))
        for i,skip64 in enumerate(self.skip64_arr[:self.depth//4]):
            x = skip64(x)
        x = self.skip64ds(x)
        for i,skip64 in enumerate(self.skip64_arr[self.depth//4:]):
            x = skip64(x)
        x = self.bn1(x)
        x = self.skip64to128(x)
        for i,skip128 in enumerate(self.skip128_arr[:self.depth//4]):
            x = skip128(x)
        x = self.bn2(x)
        x = self.skip128ds(x)
        x = x.view(-1, 8192 )
        x = self.fc_seqn(x)
        return x
```

# Outline

# Training the Multi-Instance Detector

- With YOLO-based reasoning, you have to do a lot more work in the training loop compared to what you have seen so far. Here's why:
  1. In each training image, you need to assign the bounding boxes for the different object instances to the most appropriate cell and, within that cell, to the most appropriate anchorbox.
  2. You need to reshape the output of the network into a tensor that looks like the `yolo_tensor_aug` for the training images.
  3. Obviously, you need to compute the loss as explained on Slides 105 through 109.
  4. And, every so often, you need to display the average loss and, possibly, the quality of prediction in terms of the correctness of the predicted object class labels and the bounding boxes.
  5. You can see these steps implemented in the method `run_code_for_training_multi_instance_detection()` defined for the inner class `YoloLikeDetector` in the `YOLOLogic` module. I'll review this code starting with the next slide.

# Training the Multi-Instance Detector (contd.)

- About the training code shown starting with Slide 126, note the three criteria for estimating three different components of the loss in Lines (1), (2), and (3).
- Pay attention to `idx`, the loop control parameter in Line (20). This parameter is for the different object instances in any given image. We assume a maximum of 5 objects instances in an image — as declared in Line (9). The parameter `idx` indexes into these object instances.
- In the code that follows Line (20), we estimate the height and the width, and the center of each bounding box in the annotations for the image at the input into the network. These calculations are carried out in Lines (21) through (32). In Lines (33) and (34), we calculate the center of the grid cell that is closest to the center of the bounding box. Finally, we calculate the displacement params in lines (35) and (36).

# Training the Multi-Instance Detector (contd.)

- In Line (37), we find the class label of the object instance from the `bbox_label_tensor` part of the annotations for the image in question.

- That gets us ready in Line (38) to find the aspect-ratio (AR) of the bounding box for the object instance at index `idx`. In lines (39) through (50), we use the value of AR to decide as to which of the five anchor-boxes we should assign the object instance to.

  [**In version prior to 2.1.0, my YOLO implementation was limited to a batch-size of 1. With a batch-size of 1, I simply discarded the iterations in which the input image contained unacceptable bounding boxes, i. e., say the bounding boxes with excessivly small widths that would lead to excessively large aspect ratios, or when every object class label was 13, meaning that there was no object in the image. With batches allowed in Version 2.1.0, I have dropped all such instance-specfic case reasoning. To compensate, Line (39) now tests whether AR is a nan, and, if so, it arbitrarily assigns it a value of 100.**]

- Finally, in Line (52), we have all the information we need to construct the `yolo_vector` for the object instance in question. In Line (54), we set the first element of this vector and we specify the one-hot vector for the class label of the object in Line (55).

# Training the Multi-Instance Detector (contd.)

- In Lines (56) and (57), we find the index of the cell to which the `yolo_vector` that has just been constructed can be assigned to.
- Iterating through all the object index values for `idx`, we enter the next yolo vector in the `yolo_tensor_aug` for the batch instance in Line (59). Lines (60) through (64) are for making sure that the last element of every yolo vector that does not contain an object is set to 13.
- We feed the image into the network in Line (65). The network outputs a flat tensor that is reshaped into the same shape as `yolo_tensor_aug` in Line (67).
- That sets us up to estimate the loss in lines (68) through (79) in accordance with the explanations presented earlier.
- The rest of the training code is for displaying the average loss in your terminal window every 500 iterations and also for displaying the estimated class labels for the objects detected in the images in a batch.

# Training the Multi-Instance Detector (contd.)

```
def run_code_for_training_multi_instance_detection(self, net, display_labels=False, display_images=False):

    net = net.to(self.yolo.device)                                                                                    ## (0)
    criterion1 = nn.BCELoss(reduction='sum')                                                                          ## (1)
    criterion2 = nn.MSELoss(reduction='sum')                                                                          ## (2)
    criterion3 = nn.CrossEntropyLoss(reduction='sum')                                                                 ## (3)
    optimizer = optim.SGD(net.parameters(), lr=self.yolo.learning_rate, momentum=self.yolo.momentum)                  ## (4)
    start_time = time.perf_counter()
    Loss_tally = []
    elapsed_time = 0.0
    yolo_interval = self.yolo.yolo_interval                                                                           ## (5)
    num_yolo_cells = (self.yolo.image_size[0] // yolo_interval) * (self.yolo.image_size[1] // yolo_interval)         ## (6)
    num_anchor_boxes =  5    # (height/width)   1/5  1/3  1/1  3/1  5/1                                             ## (7)
    max_obj_num  = 5                                                                                                  ## (8)
    for epoch in range(self.yolo.epochs):                                                                             ## (9)
        print("")
        running_loss = 0.0                                                                                            ## (10)
        for iter, data in enumerate(self.train_dataloader):
            im_tensor, seg_mask_tensor, bbox_tensor, bbox_label_tensor, num_objects_in_image = data                   ## (11)
            im_tensor   = im_tensor.to(self.yolo.device)
            seg_mask_tensor = seg_mask_tensor.to(self.yolo.device)
            bbox_tensor = bbox_tensor.to(self.yolo.device)
            bbox_label_tensor = bbox_label_tensor.to(self.yolo.device)
            yolo_tensor = torch.zeros( im_tensor.shape[0], num_yolo_cells, num_anchor_boxes, 8 ).to(self.yolo.device)  ## (12)
            yolo_tensor_aug = torch.zeros(im_tensor.shape[0], num_yolo_cells,num_anchor_boxes,9).to(self.yolo.device)  ## (13)
            cell_height = yolo_interval                                                                               ## (14)
            cell_width = yolo_interval                                                                                ## (15)
            num_cells_image_width = self.yolo.image_size[0] // yolo_interval                                          ## (16)
            num_cells_image_height = self.yolo.image_size[1] // yolo_interva                                          ## (17)
            ## ibx is batch instance index
            for ibx in range(im_tensor.shape[0]):                                                                     ## (18)
                ## idx is for object index
                num_of_objects =   (torch.sum(bbox_label_tensor[ibx] != 13).type(torch.uint8)).item()                 ## (19)
                for idx in range(num_of_objects):                                                                     ## (20)
                    ## Note that the bounding-box coords are in the (x,y) format, with x-positive going to
                    ## the right and y-positive going down:
                    height_center_bb =  (bbox_tensor[ibx,idx,1] + bbox_tensor[ibx,idx,3]) / 2.0                       ## (21)
                    width_center_bb =  (bbox_tensor[ibx,idx,0] + bbox_tensor[ibx,idx,2]) / 2.0                        ## (22)
                    obj_bb_height = bbox_tensor[ibx,idx,3] -  bbox_tensor[ibx,idx,1]                                  ## (23)
                    obj_bb_width = bbox_tensor[ibx,idx,2] - bbox_tensor[ibx,idx,0]                                    ## (24)
                    cell_row_indx =  (height_center_bb / yolo_interval).int()          ## for the i coordinate       ## (25)
                    cell_col_indx =  (width_center_bb / yolo_interval).int()           ## for the j coordinates      ## (26)
                    cell_row_indx = torch.clamp(cell_row_indx, max=num_cells_image_height - 1)                        ## (27)
                    cell_col_indx = torch.clamp(cell_col_indx, max=num_cells_image_width - 1)                         ## (28)
                    bh  =  obj_bb_height.float() / yolo_interval                                                      ## (29)
                    bw  =  obj_bb_width.float()  / yolo_interval                                                      ## (30)
                    ## You have to be CAREFUL about object center calculation since bounding-box coordinates
                    ## are in (x,y) format --- with x-positive going to the right and y-positive going down.
                    obj_center_x =  (bbox_tensor[ibx,idx][2].float() +  bbox_tensor[ibx,idx][0].float()) / 2.0        ## (31)
                    obj_center_y =  (bbox_tensor[ibx,idx][3].float() +  bbox_tensor[ibx,idx][1].float()) / 2.0        ## (32)
                    ## Now you need to switch back from (x,y) format to (i,j) format:
                    yolocell_center_i =  cell_row_indx*yolo_interval + float(yolo_interval) / 2.0                     ## (33)
                    yolocell_center_j =  cell_col_indx*yolo_interval + float(yolo_interval) / 2.0                     ## (34)
```



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

# Training the Multi-Instance Detector (contd.)

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

```
            del_x  =  (obj_center_x.float() - yolocell_center_j.float()) / yolo_interval                    ## (35)
            del_y  =  (obj_center_y.float() - yolocell_center_i.float()) / yolo_interval                    ## (36)
            class_label_of_object = bbox_label_tensor[ibx,idx].item()                                      ## (37)
            AR = obj_bb_height.float() / obj_bb_width.float()                                              ## (38)
            if torch.isnan(AR):                                                                            ## (39)
                AR = 100.0                                                                                 ## (40)
            else:
                AR = AR.item()                                                                             ## (41)
            if AR <= 0.2:                                                                                  ## (42)
                anch_box_index = 0                                                                         ## (43)
            elif 0.2 < AR <= 0.5:                                                                          ## (44)
                anch_box_index = 1                                                                         ## (45)
            elif 0.5 < AR <= 1.5:                                                                          ## (46)
                anch_box_index = 2                                                                         ## (47)
            elif 1.5 < AR <= 4.0:                                                                          ## (48)
                anch_box_index = 3                                                                         ## (49)
            elif AR > 4.0:                                                                                 ## (50)
                anch_box_index = 4                                                                         ## (51)
            yolo_vector = torch.FloatTensor([0,del_x, del_y, bh, bw, 0, 0, 0] ).to(self.yolo.device)       ## (52)
            if class_label_of_object != 13:                                                                ## (53)
                yolo_vector[0] = 1.0                                                                       ## (54)
                yolo_vector[5 + class_label_of_object] = 1                                                 ## (55)
            yolo_cell_index =  (cell_row_indx * num_cells_image_width  +  cell_col_indx).type(torch.uint8) ## (56)
            yolo_cell_index = yolo_cell_index.item()                                                       ## (57)
            yolo_tensor[ibx, yolo_cell_index, anch_box_index] = yolo_vector                                ## (58)
            yolo_tensor_aug[ibx, yolo_cell_index, anch_box_index,:-1] = yolo_vector                        ## (59)
##  If no obj is present, throw all the probability weight into the last element:
for ibx in range(im_tensor.shape[0]):                                                                      ## (60)
    for icx in range(num_yolo_cells):                                                                      ## (61)
        for iax in range(num_anchor_boxes):                                                                ## (62)
            if yolo_tensor_aug[ibx, icx, iax, 0] == 0:                                                     ## (63)
                yolo_tensor_aug[ibx, icx, iax,-1] = 1                                                      ## (64)
optimizer.zero_grad()                                                                                      ## (65)
output = net(im_tensor)                                                                                    ## (66)
predictions_aug = output.view(-1,num_yolo_cells,num_anchor_boxes,9)                                        ## (67)

##  Now that we have the output of the network, must calculate the loss.  We initialize the loss tensors
##  for this iteration of training:
loss = torch.tensor(0.0, requires_grad=True).float().to(self.yolo.device)                                  ## (68)
##  Estimating presence/absence of object with the Binary Cross Entropy loss:
bceloss = criterion1( nn.Sigmoid()(predictions_aug[:,:,:,0]), yolo_tensor_aug[:,:,:,0] )                   ## (69)
loss += bceloss                                                                                            ## (70)
## MSE loss for the regression params for the object bounding boxes:
regression_loss = criterion2(predictions_aug[:,:,:,1:5], yolo_tensor_aug[:,:,:,1:5])                       ## (71)
loss += regression_loss                                                                                    ## (72)
##  CrossEntropy loss for object class labels:
targets = yolo_tensor_aug[:,:,:,5:]                                                                        ## (73)
targets = targets.view(-1,4)                                                                               ## (74)
targets = torch.argmax(targets, dim=1)                                                                     ## (75)
probs = predictions_aug[:,:,:,5:]                                                                          ## (76)
probs = probs.view(-1,4)                                                                                   ## (77)
```



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

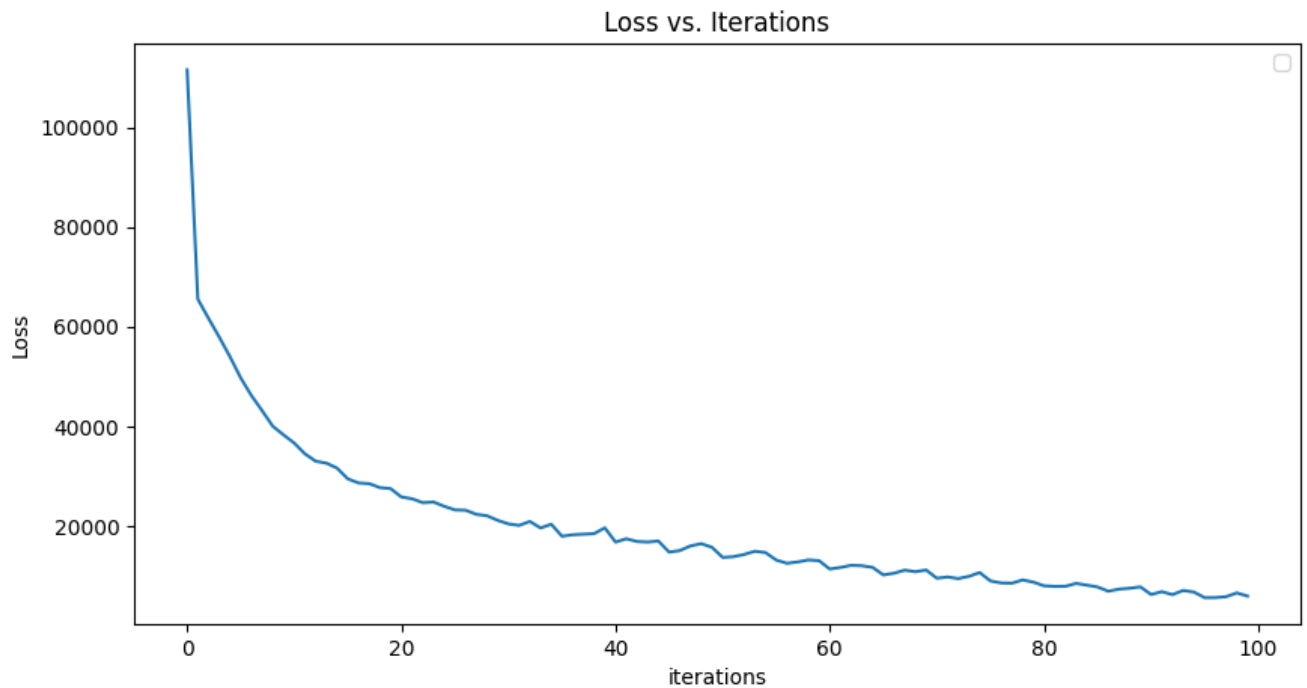
# Training the Multi-Instance Detector (contd.)

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

```
class_labeling_loss = criterion3( probs, targets)                                                    ## (78)
loss +=  class_labeling_loss                                                                         ## (79)
loss.backward()                                                                                      ## (80)
optimizer.step()                                                                                     ## (81)
running_loss += loss.item()                                                                          ## (82)
if iter%500==499:                                                                                    ## (83)
    current_time = time.perf_counter()
    elapsed_time = current_time - start_time
    avg_loss = running_loss / float(500)                                                             ## (84)
    print("\n[epoch:%d/%d, iter=%4d  elapsed_time=%5d secs]      mean value for loss: %7.4f" %
                                  (epoch+1,self.yolo.epochs, iter+1, elapsed_time, avg_loss))        ## (85)
    Loss_tally.append(running_loss)
    FILE1.write("%.3f\n" % avg_loss)
    FILE1.flush()
    running_loss = 0.0                                                                               ## (86)
    running_bceloss = 0.0
    running_regressionloss = 0.0
    running_labelingloss = 0.0
    if display_labels:
        for ibx in range(predictions_aug.shape[0]):                          # for each batch image  ## (87)
            icx_2_best_anchor_box = {ic : None for ic in range(36)}                                  ## (88)
            for icx in range(predictions_aug.shape[1]):                      # for each yolo cell    ## (89)
                cell_predi = predictions_aug[ibx,icx]                                                ## (90)
                prev_best = 0                                                                        ## (91)
                for anchor_bdx in range(cell_predi.shape[0]):                                        ## (92)
                    if cell_predi[anchor_bdx][0] > cell_predi[prev_best][0]:                         ## (93)
                        prev_best = anchor_bdx                                                       ## (94)
                best_anchor_box_icx = prev_best                                                      ## (95)
                icx_2_best_anchor_box[icx] = best_anchor_box_icx                                     ## (96)
            sorted_icx_to_box = sorted(icx_2_best_anchor_box,                                        ## (97)
                  key=lambda x: predictions_aug[ibx,x,icx_2_best_anchor_box[x]][0].item(), reverse=True)  ## (98)
            retained_cells = sorted_icx_to_box[:5]                                                   ## (99)
            objects_detected = []                                                                    ## (100)
            for icx in retained_cells:                                                               ## (101)
                pred_vec = predictions_aug[ibx,icx, icx_2_best_anchor_box[icx]]                      ## (102)
                class_labels_predi  = pred_vec[-4:]                                                  ## (103)
                class_labels_probs = torch.nn.Softmax(dim=0)(class_labels_predi)                     ## (104)
                class_labels_probs = class_labels_probs[:-1]                                         ## (105)
                ##  The threshold of 0.25 applies only to the case of there being 3 classes of objects
                ##  in the dataset.  In the absence of an object, the values in the first three nodes
                ##  that represent the classes should all be less than 0.25. In general, for N classes
                ##  you would want to set this threshold to 1.0/N
                if torch.all(class_labels_probs < 0.25):                                             ## (115)
                    predicted_class_label = None                                                     ## (116)
                else:
                    best_predicted_class_index = (class_labels_probs == class_labels_probs.max())    ## (117)
                    best_predicted_class_index =torch.nonzero(best_predicted_class_index,as_tuple=True)  ## (118)
                    predicted_class_label =self.yolo.class_labels[best_predicted_class_index[0].item()]  ## (119)
                    objects_detected.append(predicted_class_label)                                   ## (120)

            print("[batch image=%d]  objects found in descending probability order: " % ibx,
                                                                    objects_detected)                ## (121)
```

# Training Loss versus Iterations

- Shown below is the training loss I get when I execute the code in `run_code_for_training_multi_instance_detection()` over 20 epochs.
- Starting with Slide 137, I will present equally impressive results on the test dataset that contains images not seen during training.



# Outline

# Challenges in Evaluating a Multi-Instance Detector

- It is straightforward to write the evaluation logic for the case of single-instance object detectors discussed earlier in this lecture. If we assume that each image has at most one instance of an object in it, then the evaluation consists of comparing the class label predicted for the image with its true label. If the detection also involves estimating the bounding-box for the object in the image, it is just as easy to evaluate the accuracy achieved for the predicted bounding box vis-a-vis the ground-truth bounding box.
- However, evaluating a multi-instance object detector is much more complicated for several reasons: First and foremost, a successful detection of an object in a given cell/anchorbox combo depends on the value predicted for the first element of the yolo-vector for that combo and also for the values estimated for the last 3 elements of the yolo-vector (the last 4 elements of the augmented yolo vector) where we estimate the probability of the class label for the object (assuming that the first element says that it is there).

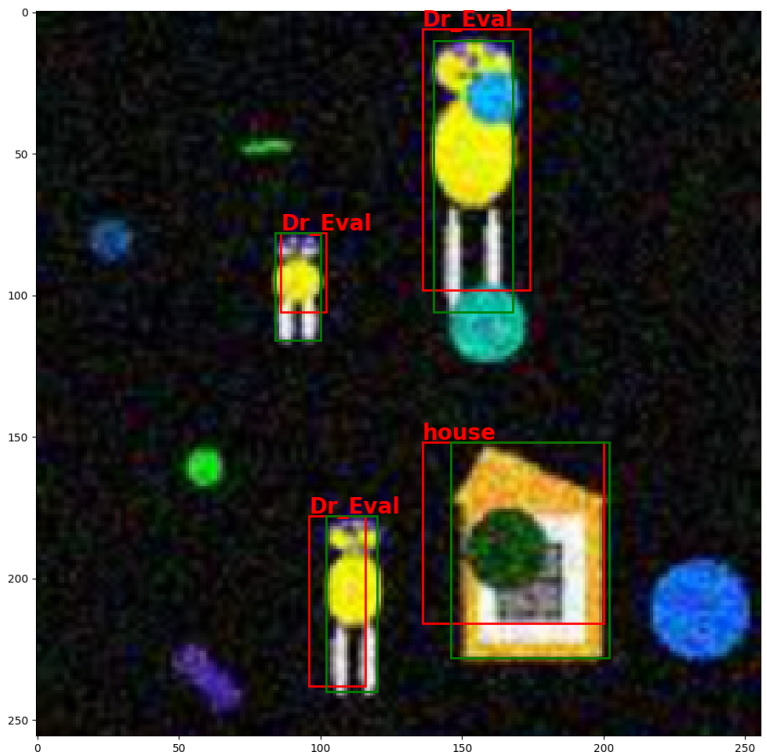
# Challenges in Evaluating a Multi-Instance Detector (contd.)

- Assuming that the detection logic applied to the $1^{st}$ and the $6^{th}$ through $8^{th}$ elements of the yolo-vector indicated the presence of an object instance, you then have to get hold of the predicted bounding box based on the values in the $2^{nd}$ through the $5^{th}$ elements of the yolo-vector and estimate the IoU metric when applied to ground-truth and the predicted bounding-boxes.

- As you will recall from the multi-loss function presented on Slides 105 through 109, we use the `nn.BCELoss` for the first element of the predicted yolo-vector for each cell/anchorbox combo, implying that we treat the predicted value in the first element as binary detector for determining the presence or absence of an object instance. Therefore, at inference time, we must apply a decision threshold to this element in order to accept the predicted instance offered. This threshold obviously has a significant bearing on the overall performance of the detector.

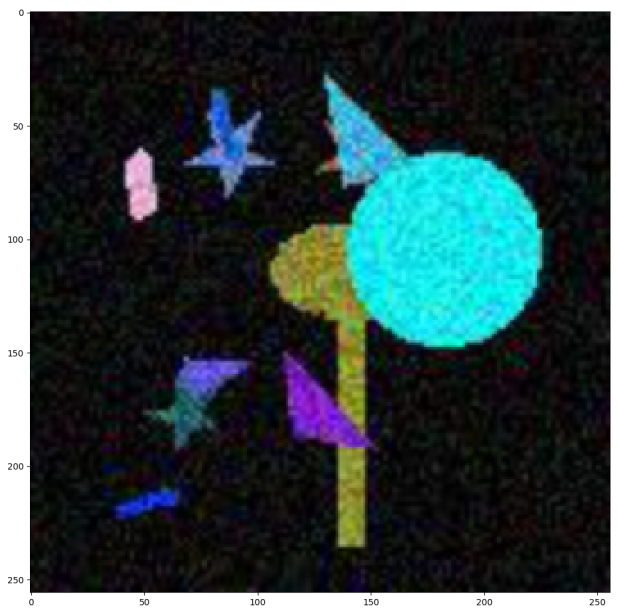
# Challenges in Evaluating a Multi-Instance Detector (contd.)

- Starting with this slide, I'll now use some results from the YOLO multi-instance object detector to illustrate the challenges involved in the evaluation of such detectors. But, first, shown below is a typical case in which all the instances are detected correctly and with the predicted bounding boxes that appear to be reasonably good approximations to the ground-truth bounding boxes.



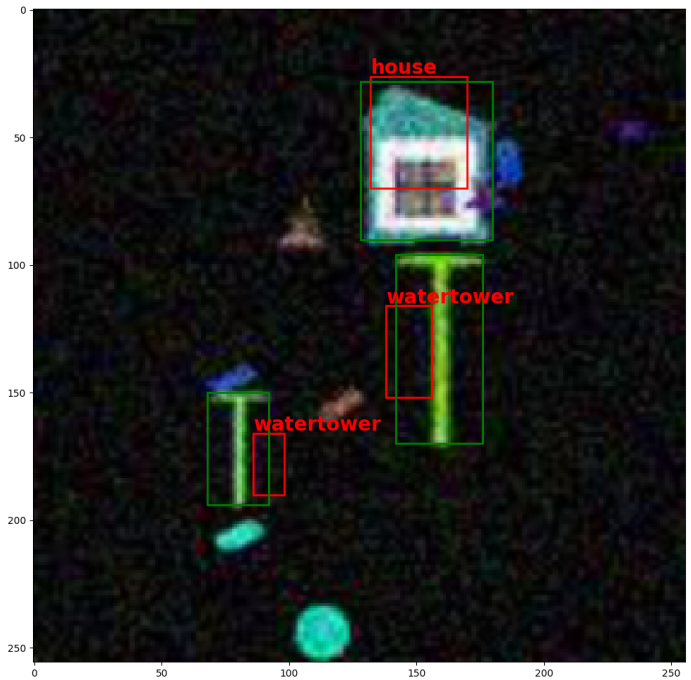
# Evaluation Issue — Missed Detections

- Now for the more difficult cases, shown below is a case in which there is a single instance of an object, watertower, but the object was not detected. The detection was missed probably because of a large degree of structured clutter over the object and in its immediate vicinity. To account for such cases in the output, the evaluation must include a category called "missed detections" for each object type.



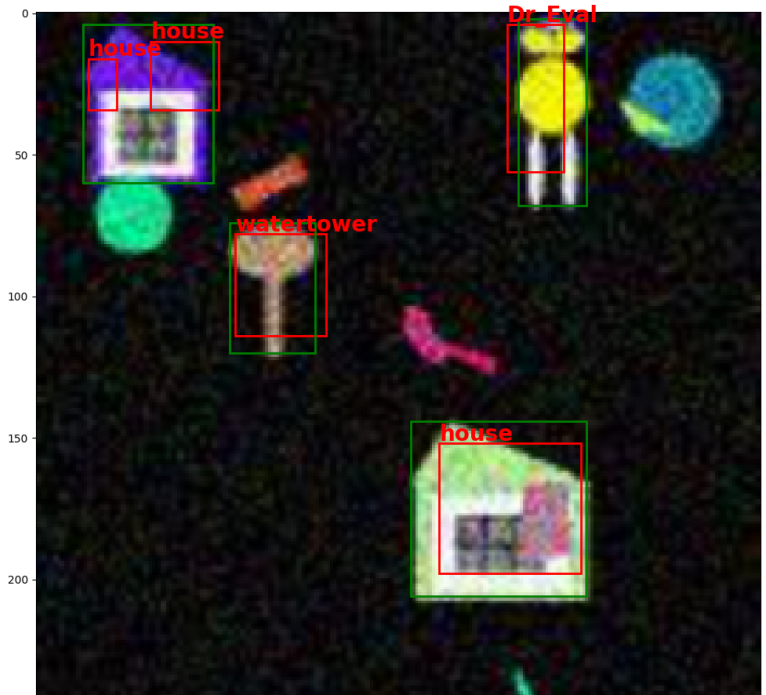
# Evaluation Issue — Inconsistent Evidence

- The example presented below shows ZERO overlap between the ground-truth bounding boxes and the predicted bounding boxes for the two watertower objects in the scene. That raises an interesting question: How to deal with the case when the labeling evidence is inconsistent with the regression evidence? That is, when the predicted bounding-box does not capture any part of the instance corresponding to the predicted label.



# Evaluation Issue — Double Detections

- Shown below is an evaluation difficulty caused by a double detection of the same object, in this case the house at the upper left. This can happen when the center of the ground-truth bounding-box for an object happens to fall exactly on the edge between two adjacent cells in the YOLO grid overlaid on the image. [Such double detections can be eliminated by the "non-maximum suppression" algorithm that uses an appropriate criterion to, first, decide that the two neighboring detections are for the same object and then rejecting the weaker of the two.]



# Outline

# Testing the Multi-Instance Object Detector

- The explanations in this section are based on the code in the function `run_code_for_testing_multi_instance_detection()` of the YOLOLogic module.
- I also assume that the function named above will be run on the test dataset that comes for experiments with multi-instance object detection using the YOLOLogic module. This dataset is NOT used for the training routine presented earlier.
- Starting with Slide 148, in the code shown in this section for `run_code_for_testing_multi_instance_detection()`, in the statements that immediately follow Line (1), we have the initializations for what is needed for the calculation of the confusion matrix for the object instance labels. And in Line (2), we do the same for the calculation of the IoU metric for measuring the accuracy of the predicted bounded boxes vis-a-vis the ground-truth bounding boxes.

# Testing the Multi-Instance Detector (contd.)

- In the statement in Line (3), the `tensor bbox_label_tensor` looks like `tensor([0,0,13,13,13], device='cuda:0',dtype=torch.uint8)` where '0' is a genuine class label for 'Dr.Eval' and the number 13 as a label represents the case when there is no object. As you will recall, each image has a max of 5 object instances in it. So the 5 positions in the tensor are for each of those instances. The bounding-boxes for each of those five instances are in the tensor `bbox_tensor` and their segmentation masks in the tensor `seg_mask_tensor`.
- In Line (4), we feed the testing data tensor into our neural network that was described earlier in Slides 102 and 103. And in Line (5), we reshape the output of the network so that it corresponds to the shape of the augmented `yolo_tensor_aug` as defined on Slide 103.
- Starting in Line (6), our goal is to look through all the cells and identify at most **five** of the cells/anchorbox combos for which **the value in the first elements of the predicted yolo-vectors are the highest**.

# Testing the Multi-Instance Detector (contd.)

- Starting in Line (7), for each of the 36 cells of the yolo-grid overlaid on the image, we examine the value in the first element of the yolo-vector for each of the five anchorboxes for the cell. For each cell, we retain the anchorbox that has the highest probability of detection in the first element of the yolo-vector. In the 'for' loop that begins in Line (6), I use `ibx` for the batch instance index. However, note that the yolo testing code shown is guaranteed to work for a batch size of just 1 since that is commonly the case at inference time. Additionally, I use `icx` for the cell index, and `anchor_bdx` as the anchorbox index for each cell.
- In Line (15), we sort all the cell/anchorbox combos retained in the loop described above and then in Line (16), we retain the top five of those on the basis of the probabilities in the first element in the predicted yolo-vectors.
- That takes to the loop in Lines (18) through (42), which is explained on the next slide.

# Testing the Multi-Instance Detector (contd.)

- In Lines (19) through (24), our first job is to figure out if the probability mass in the $6^{th}$, $7^{th}$, and $8^{th}$ elements of the predicted yolo-vector is large enough to warrant paying further attention to the cell/anchorbox combo in question.
- To understand the logic in Lines (22), (23) and (24), you need to recall that an additional element was appended to the 8-element yolo-vectors for accumulating the probability mass for the case when there is **no** object instance assigned to a cell/anchorbox combo.
- In Line (22) we get hold of the probabilities in the $6^{th}$, $7^{th}$, and $8^{th}$ elements of the augmented yolo-vector. Since we have only three object classes in our training data, a noise-induced random assignment of probabilities to the $6^{th}$, $7^{th}$, and $8^{th}$ elements and the additional $9^{th}$ element used for augmentation will roughly equal 0.25.

# Testing the Multi-Instance Detector (contd.)

- Therefore, if the largest value stored in the elements $6^{th}$, $7^{th}$, and $8^{th}$ does not exceed, say, 0.2, we can be reasonably certain that the cell/anchorbox in question does not contain a viable instance.

- Lines (25) through (42) deal with the case when we are reasonably certain that we are looking at a cell/anchorbox combo that contains an object instance. Of these, Lines (25) through (29) get hold of the element index that has the largest value amongst the $6^{th}$, $7^{th}$, and $8^{th}$ elements. The statement in Line (25) returns a Boolean tensor whose elements are either True or False. With the statements in Line (26) and (27), we get hold of the index of the element where the value is True. That gives us the index of the class label for the object instance. Line (28) gives us the symbolic name of the class.

- Lines (30) through (42) are for getting hold of the predictions for the regression elements of the yolo-vectors as explained on the next slide.

# Testing the Multi-Instance Detector (contd.)

- In Line (30), we store the $2^{nd}$ through $5^{th}$ elements of the predicted yolo-vector in the variable `pred_regression_vec`. The first two elements of this vector, as accessed in Line (31), are for the $\delta x$ and $\delta y$ displacements of the center of the predicted bounding-box vis-a-vis the center of the cell as shown on Slide 95. And the other two elements as accessed in Line (32) are for the height $h$ and the width $w$ of the bounding box.
- Since the variables $\delta x$, $\delta y$, $h$, and $w$ are normalized by the variable `yolo_interval` at training time, we de-normalize them and use them to estimate the coordinates of the actual center of the predicted bounding box and its top-left corner in Lines (33) through (42).
- Just after Line (42), the ground-truth bounding-boxes are in the $(x1, y1, x2, y2)$ format in which $(x1, y1)$ are the coordinates of the top-left corner and $(x2, y2)$ of the bottom-right corner. On the other hand, the predicted bounding boxes are in the $(x, y, h, w)$ format.

# Testing the Multi-Instance Detector (contd.)

- Note that when I use the notation $(x, y)$ for the pixel coordinates, $x$ is the horizontal coordinate that is positive to the right and $y$ is the vertical coordinate positive downwards.
- The goal of the loop in Lines (45) through (56) is to compare each of the predicted bounding boxes with every one of the ground-truth bounding boxes to figure out the mappings between the predicted bboxes and the ground-truth bboxes. This comparison involves calculating the IoU distance between a predicted bbox and a gt bbox by calling `IoU_calculator(predicted_bboxes[i], gt_bboxes[j])` in Line (54).

  [**An important consequence of this IoU based comparison: If there is no intersection between the predicted and the ground-truth bounding boxes and even if the classification label was correctly identified, that would be treated as a missed detection.**]

- For each predicted bounding box, we rank order the ground-truth bounding boxes on the basis of the IoU values and choose the best ground-truth bounding-box in Line (56).

# Testing the Multi-Instance Detector (contd.)

- Note that the predicted to the ground-truth bbox mappings stored in the `mapping_from_pred_to_gt` dictionary are in terms of the integer indexes associated with the bounding boxes.

- Now that we have found found the best possible pairings between the predicted and the gt bboxes, we need to compare their class labels for filling out the `confusion_matrix` that was defined originally in the statement just after Line (1). This comparison is based on the integer indexes associated with the symbolic class names.

- The integer indexes for the symbolic class labels of the ground-truth bounding-boxes are stored in the variable `gt_labels` in Line (57) and the same for the predicted bounding-boxes in `pred_labels_ints` in Line (58). In Line (58), we accumulated the integer integer indexes for the predicted class labels in the variable `predicted_label_index_vals` earlier in Line (27).

# Testing the Multi-Instance Detector (contd.)

- Using the integer indexes for the class labels and the mapping from the integer indexes associated with the predicted bboxes to the ground-truth bboxes, we now fill out the `confusion_matrix` in Line (60).
- The statement in Line (67) updates the IoU scores accumulated for each class label. To understand this statement, note that the dictionary `mapping_from_pred_to_gt` constructed in Line (56) records not only the ground-truth bbox index for a given predicted bbox index but also the IoU score for the mapping. In Line (56), the target index and the IoU score are put away as a tuple.
- The code in Lines (68) through (71) is for displaying the image that also shows the annotated bounding-boxes for the predicted objects instances. And the code segment that begins with Line (72) is for displaying the Confusion Matrix and other ancillary stats related to the classification performance of the detector. The section of the code that begins with Line (73) outputs the IoU scores for the different classes.

# Testing the Multi-Instance Detector (contd.)

- Finally, the definition of the `IoU_calculator` is provided starting at Line (74).
- The `IoU_calculator` has two required arguments: the two bounding-boxes whose IoU you want to calculate. It assumes that each bounding-box is described by a 4-tuple whose first two entries are the integer coordinates of the top-left corner of the bounding box and the last two entries the same for the bottom-right corner.
- The corner coordinates in the bounding-box args supplied to the `IoU_calculator` are in the $(x, y)$ format as opposed to the $(i, j)$ format. In the $(x, y)$ format, $x$ is the horizontal coordinate with positive going to the right and $y$ the vertical coordinate with positive pointing straight down.

# Testing the Multi-Instance Detector (contd.)

```
def run_code_for_testing_multi_instance_detection(self, net, display_images=False):
    yolo_debug = False
    net.load_state_dict(torch.load(self.yolo.path_saved_yolo_model))
    net = net.to(self.yolo.device)
    yolo_interval = self.yolo.yolo_interval
    num_yolo_cells = (self.yolo.image_size[0] // yolo_interval) * (self.yolo.image_size[1] // yolo_interval)
    num_anchor_boxes =  5    # (height/width)   1/5  1/3  1/1  3/1  5/1
    ##  The next 5 assignment are for the calculations of the confusion matrix:
    confusion_matrix = torch.zeros(3,3)   #  We have only 3 classes:  Dr. Eval, house, and watertower                    ## (1)
    class_correct = [0] * len(self.yolo.class_labels)
    class_total = [0] * len(self.yolo.class_labels)
    totals_for_conf_mat = 0
    totals_correct = 0
    ##  We also need to report the IoU values for the different types of objects
    iou_scores = [0] * len(self.yolo.class_labels)                                                                      ## (2)
    with torch.no_grad():
        for iter, data in enumerate(self.test_dataloader):
            im_tensor, seg_mask_tensor, bbox_tensor, bbox_label_tensor, num_objects_in_image = data                     ## (3)
            if iter % 50 == 49:
                im_tensor   = im_tensor.to(self.yolo.device)
                seg_mask_tensor = seg_mask_tensor.to(self.yolo.device)
                bbox_tensor = bbox_tensor.to(self.yolo.device)
                output = net(im_tensor)                                                                                 ## (4)
                predictions = output.view(self.yolo.batch_size,num_yolo_cells,num_anchor_boxes,9)                       ## (5)
                for ibx in range(predictions.shape[0]):                                 # for each batch image
                    icx_2_best_anchor_box = {ic : None for ic in range(36)}                                             ## (6)
                    for icx in range(predictions.shape[1]):                             # for each yolo cell            ## (7)
                        cell_predi = predictions[ibx,icx]                                                               ## (8)
                        prev_best = 0                                                                                   ## (9)
                        for anchor_bdx in range(cell_predi.shape[0]):                                                   ## (10)
                            if cell_predi[anchor_bdx][0] > cell_predi[prev_best][0]:                                    ## (11)
                                prev_best = anchor_bdx                                                              ## (12)
                        best_anchor_box_icx = prev_best                                                                 ## (13)
                        icx_2_best_anchor_box[icx] = best_anchor_box_icx                                                ## (14)
                    sorted_icx_to_box = sorted(icx_2_best_anchor_box,     \
                                key=lambda x: predictions[ibx,x,icx_2_best_anchor_box[x]][0].item(), reverse=True)      ## (15)
                    retained_cells = sorted_icx_to_box[:5]                                                              ## (16)
                ## We will now identify the objects in the retained cells and also extract their bounding boxes:
                objects_detected = []                                                                                   ## (17)
                predicted_bboxes  = []
                predicted_labels_for_bboxes = []
                predicted_label_index_vals = []
```

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

# Evaluation Logic (contd.)

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

```
for icx in retained_cells:                                                                                    ## (18)
    pred_vec = predictions[ibx,icx, icx_2_best_anchor_box[icx]]                                               ## (19)
    class_labels_predi  = pred_vec[-4:]                                                                       ## (20)
    class_labels_probs = torch.nn.Softmax(dim=0)(class_labels_predi)                                          ## (21)
    class_labels_probs = class_labels_probs[:-1]                                                              ## (22)
    if torch.all(class_labels_probs < 0.2):                                                                   ## (23)
        predicted_class_label = None                                                                          ## (24)
    else:
        ## Get the predicted class label:
        best_predicted_class_index = (class_labels_probs == class_labels_probs.max())                         ## (25)
        best_predicted_class_index = torch.nonzero(best_predicted_class_index, as_tuple=True)                 ## (26)
        predicted_label_index_vals.append(best_predicted_class_index[0].item())                               ## (27)
        predicted_class_label = self.yolo.class_labels[best_predicted_class_index[0].item()]                   ## (28)
        predicted_labels_for_bboxes.append(predicted_class_label)                                             ## (29)
        ## Analyze the predicted regression elements:
        pred_regression_vec = pred_vec[1:5].cpu()                                                             ## (30)
        del_x,del_y = pred_regression_vec[0], pred_regression_vec[1]                                          ## (31)
        h,w = pred_regression_vec[2], pred_regression_vec[3]                                                  ## (32)
        h *= yolo_interval                                                                                    ## (33)
        w *= yolo_interval                                                                                    ## (34)
        cell_row_index =  icx // 6                                                                            ## (35)
        cell_col_index =  icx % 6                                                                             ## (37)
        bb_center_x = cell_col_index * yolo_interval  +  yolo_interval/2  +  del_x * yolo_interval            ## (38)
        bb_center_y = cell_row_index * yolo_interval  +  yolo_interval/2  +  del_y * yolo_interval            ## (39)
        bb_top_left_x =  int(bb_center_x - w / 2.0)                                                           ## (40)
        bb_top_left_y =  int(bb_center_y - h / 2.0)                                                           ## (41)
        predicted_bboxes.append( [bb_top_left_x, bb_top_left_y, int(w), int(h)] )                             ## (42)
## make a deep copy of the predicted_bboxes for eventual visual display:
saved_predicted_bboxes =[ predicted_bboxes[i][:] for i in range(len(predicted_bboxes)) ]                      ## (43)
##  To account for the batch axis, the bb_tensor is of shape [1,5,4]. At this point, we get rid of the batch axis
##  and turn the tensor into a numpy array.
gt_bboxes = torch.squeeze(bbox_tensor).cpu().numpy()                                                          ## (44)
##  We want the predictions bboxes to be in the same format as the GT bboxes:
for pred_bbox in predicted_bboxes:                                                                            ## (45)
    w,h = pred_bbox[2], pred_bbox[3]                                                                          ## (46)
    pred_bbox[2] = pred_bbox[0] + w                                                                           ## (47)
    pred_bbox[3] = pred_bbox[1] + h                                                                           ## (48)
## These are the mappings from indexes for the predicted bboxes to the indexes for the gt bboxes:
mapping_from_pred_to_gt = { i : None for i in range(len(predicted_bboxes))}                                   ## (49)
for i in range(len(predicted_bboxes)):                                                                        ## (50)
    gt_possibles = {k : 0.0 for k in range(5)}      ## 0.0 for IoU                                           ## (51)
    for j in range(len(gt_bboxes)):                                                                           ## (52)
        if all(gt_bboxes[j][x] == 0 for x in range(4)): continue      ## 4 is for the four coords of a bbox  ## (53)
        gt_possibles[j] = self.IoU_calculator(predicted_bboxes[i], gt_bboxes[j])                              ## (54)
    sorted_gt_possibles =  sorted(gt_possibles, key=lambda x: gt_possibles[x], reverse=True)                  ## (55)
    mapping_from_pred_to_gt[i] = (sorted_gt_possibles[0], gt_possibles[sorted_gt_possibles[0]])               ## (56)
gt_labels = torch.squeeze(bbox_label_tensor).cpu().numpy()                                                    ## (57)
## These are the predicted numeric class labels for the predicted bboxes in the image
pred_labels_ints = predicted_label_index_vals                                                                 ## (58)
```



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

# Evaluation Logic (contd.)

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

```
            for i,bbox_pred in enumerate(predicted_bboxes):                                                                    ## (59)
                if gt_labels[pred_labels_ints[i]] != 13:                                                                       ## (60)
                    confusion_matrix[gt_labels[mapping_from_pred_to_gt[i][0]]][pred_labels_ints[i]]  +=  1                     ## (61)
                totals_for_conf_mat += 1                                                                                       ## (62)
                class_total[gt_labels[mapping_from_pred_to_gt[i][0]]] += 1                                                     ## (63)
                if gt_labels[mapping_from_pred_to_gt[i][0]] == pred_labels_ints[i]:                                            ## (64)
                    totals_correct += 1                                                                                        ## (65)
                    class_correct[gt_labels[mapping_from_pred_to_gt[i][0]]] += 1                                               ## (66)
                iou_scores[gt_labels[mapping_from_pred_to_gt[i][0]]] += mapping_from_pred_to_gt[i][1]                          ## (67)
            ## If the user wants to see the image with the predicted bboxes and also the predicted labels:
            if display_images:
                predicted_bboxes = saved_predicted_bboxes                                                                      ## (68)
                if yolo_debug:
                    print("[batch image=%d]  objects found in descending probability order: " % ibx, predicted_labels_for_bboxes)
                logger = logging.getLogger()
                old_level = logger.level
                logger.setLevel(100)
                fig = plt.figure(figsize=[12,12])
                ax = fig.add_subplot(111)
                display_scale = 2
                new_im_tensor = torch.nn.functional.interpolate(im_tensor,
                                                        scale_factor=display_scale, mode='bilinear', align_corners=False)       ## (69)
                ax.imshow(np.transpose(torchvision.utils.make_grid(new_im_tensor,
                                                        normalize=True, padding=3, pad_value=255).cpu(), (1,2,0)))              ## (70)
                for i,bbox_pred in enumerate(predicted_bboxes):                                                                ## (71)
                    x,y,w,h = np.array(bbox_pred)
                    x,y,w,h = [item * display_scale for item in (x,y,w,h)]
                    rect = Rectangle((x,y),w,h,angle=0.0,edgecolor='r',fill = False,lw=2)
                    ax.add_patch(rect)
                    ax.annotate(predicted_labels_for_bboxes[i], (x,y-1), color='red', weight='bold', fontsize=10*display_scale)
                plt.show()
                logger.setLevel(old_level)

        ##  Our next job is to present to the user the information collected for the confusion matrix for the validation dataset:
        for j in range(len(self.yolo.class_labels)):                                                                           ## (72)
            print('Prediction accuracy for %5s : %2d %%' % (self.yolo.class_labels[j], 100 * class_correct[j] / class_total[j]))
        print("\n\n\nOverall accuracy of the network on the 10000 test images: %d %%" % (100 * sum(class_correct) / float(sum(class_total))))
        print("\n\nDisplaying the confusion matrix:\n")
        out_str = "                "
        for j in range(len(self.yolo.class_labels)):  out_str +=  "%15s" % self.yolo.class_labels[j]
        out_str +=  "%15s" % "missing"
        print(out_str + "\n")
        for i,label in enumerate(self.yolo.class_labels):
            out_percents = [100 * confusion_matrix[i,j] / float(class_total[i]) for j in range(len(self.yolo.class_labels))]
            missing_percent = 100 - sum(out_percents)
            out_percents.append(missing_percent)
            out_percents = ["%.2f" % item.item() for item in out_percents]
            out_str = "%10s:  " % self.yolo.class_labels[i]
            for j in range(len(self.yolo.class_labels)+1): out_str +=  "%15s" % out_percents[j]
            print(out_str)
```



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

# Evaluation Logic (contd.)

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

```
    print("\n\nNOTE 1: 'missing' means that an object instance of that label was NOT extracted from the image.")
    print("\nNOTE 2: 'prediction accurary' means the labeling accuracy for the extracted objects.")
    print("\nNOTE 3: True labels are in the left-most column and the predicted labels at the top of the table.")

    ##  Finally, we present to the user the IoU scores for each of the object types:                                    ## (73)
    iou_score_by_label = {self.yolo.class_labels[i] : 0.0 for i in range(len(self.yolo.class_labels))}
    for i,label in enumerate(self.yolo.class_labels):
       iou_score_by_label[self.yolo.class_labels[i]] = iou_scores[i]/float(class_total[i])
    print("\n\nIoU scores for the different types of objects: ")
    for obj_type in iou_score_by_label:
        print("\n    %10s:    %.4f" % (obj_type, iou_score_by_label[obj_type]))


def IoU_calculator(self, bbox1, bbox2, seg_mask1=None, seg_mask2=None):                                              ## (74)
    """
    I assume that a bbox is defined by a 4-tuple, with the first two integers standing for the
    top-left coordinate in the (x,y) format and the last two integers for the bottom-right coordinates
    in also the (x,y) format.  By (x,y) format I mean that x stands for the horiz axis with positive to
    the right and y for the vert coord with positive pointing downwards.
    """
    union = intersection = 0
    b1x1,b1y1,b1x2,b1y2 = bbox1                              ## b1 refers to bbox1
    b2x1,b2y1,b2x2,b2y2 = bbox2                              ## b2 refers to bbox2
    for x in range(self.yolo.image_size[0]):                  ## image is 128x128
        for y in range(self.yolo.image_size[1]):
            if  ( ( ( (x >= b1x1) and (x >= b2x1) ) and  ( (y >= b1y1) and (y >= b2y1) ) )  and  \
                ( ( (x < b1x2) and (x < b2x2) )  and  ((y < b1y2)  and (y < b2y2)) ) ):
                intersection += 1
            if  ( ( (x >= b1x1) and (x <b1x2) ) and  ((y >= b1y1) and (y < b1y2)) ):
                union += 1
            if  ( ( (x >= b2x1) and (x <b2x2) ) and  ((y >= b2y1) and (y < b2y2)) ):
                union += 1
    union = union - intersection
    if union == 0.0:
        raise Exception("something_wrong")
    iou = intersection / float(union)
    return iou
```

# Outline

# Evaluation Results on Unseen Images

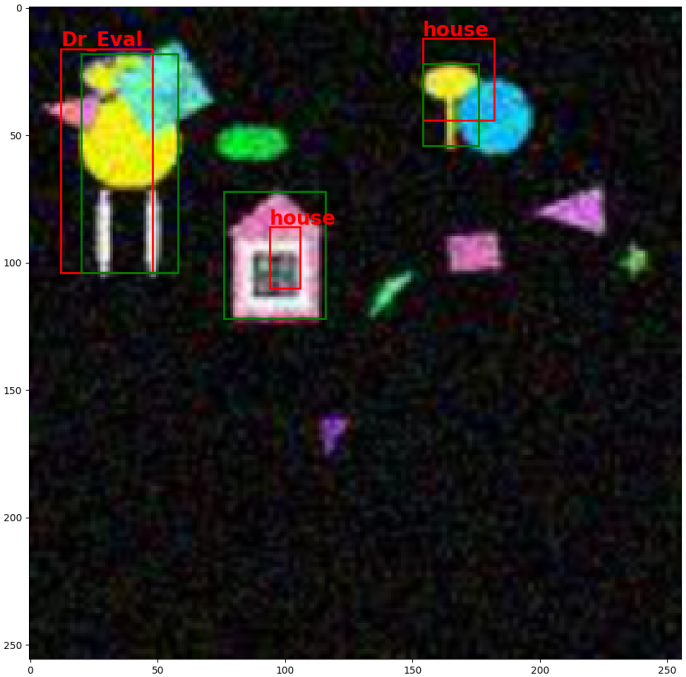
- I'll now show some results from running the script

  ```
  multi_instance_object_detection.py
  ```

  in the `ExamplesObjectDetection` directory of the YOLOLogic module. In order to get the sort of results I show in this section, please make sure that your version of the YOLOLogic is 2.0.8 or higher.

- The results I'll be showing are based on 20 epochs of training. When doing their multi-instance object detection homework based on the code illustrated in these lecture slides, some students in my class at Purdue have run their training for up to 80 epochs. I have noticed that it can take a lot more than 20 epochs of training for getting the best possible results — especially on datasets as complex as the COCO dataset.

# Evaluation Results on Unseen Images (contd.)



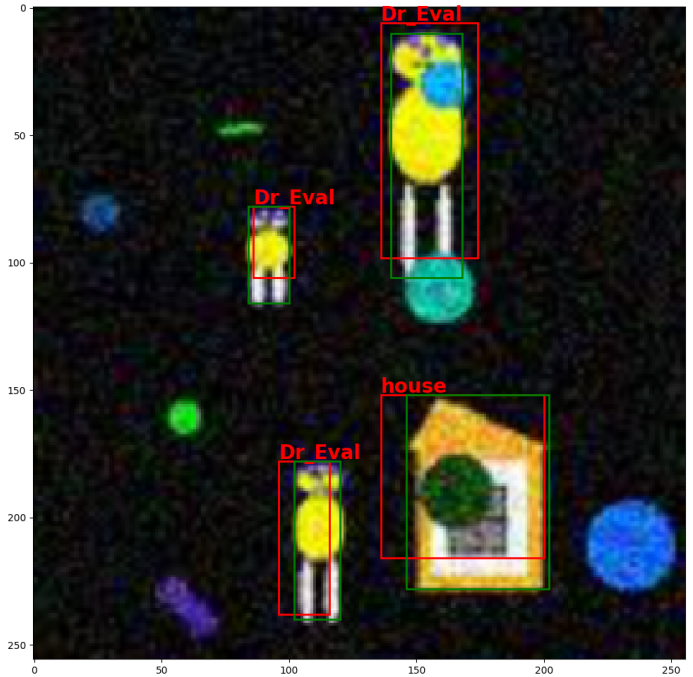

## (a) At iteration 100

Showing output for test image 100:

For predicted bbox 0: the best gt bbox is: 1
For predicted bbox 1: the best gt bbox is: 0
For predicted bbox 2: the best gt bbox is: 2

mapping_from_pred_to_gt: {0: (1, 0.48), 1: (0, 0.20), 2: (2, 0.20)}

for i=0, the predicted label: house the ground_truth label: house
for i=1, the predicted label: Dr_Eval the ground_truth label: Dr_Eval
for i=2, the predicted label: watertower the ground_truth label: watertower




## (b) At iteration 200

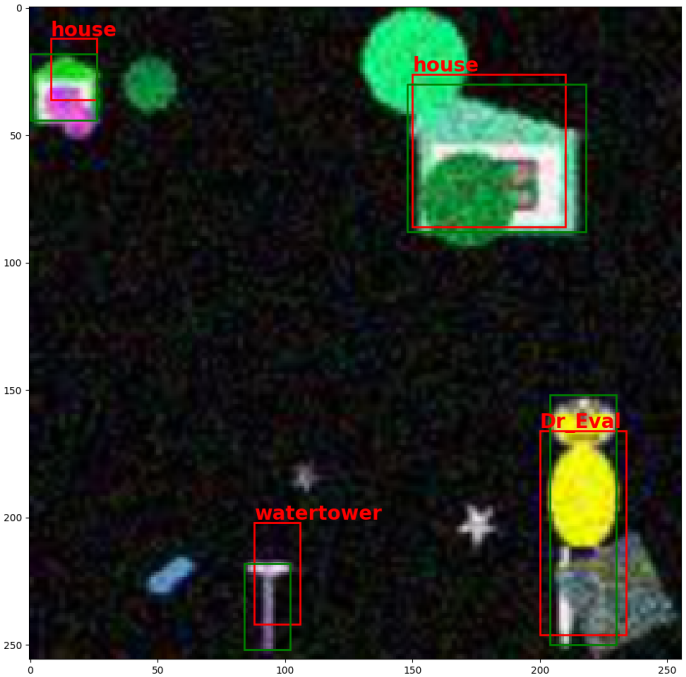
Showing output for test image 200:

For predicted bbox 0: the best gt bbox is: 2
For predicted bbox 1: the best gt bbox is: 0
For predicted bbox 2: the best gt bbox is: 3
For predicted bbox 3: the best gt bbox is: 1

mapping_from_pred_to_gt: {0: (2, 0.61), 1: (0, 0.81), 2: (3, 0.43), 3: (1, 0.3

for i=0, the predicted label: house the ground_truth label: house
for i=1, the predicted label: Dr_Eval the ground_truth label: Dr_Eval
for i=2, the predicted label: Dr_Eval the ground_truth label: Dr_Eval
for i=3, the predicted label: Dr_Eval the ground_truth label: Dr_Eval

# Evaluation Results on Unseen Images (contd.)



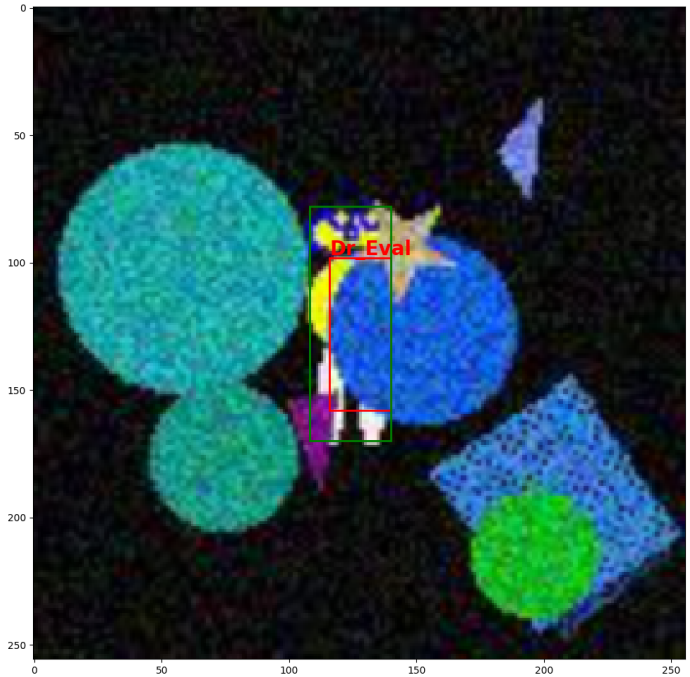

## (a) At iteration 300

Showing output for test image 300:

For predicted bbox 0: the best gt bbox is: 0
For predicted bbox 1: the best gt bbox is: 3
For predicted bbox 2: the best gt bbox is: 1
For predicted bbox 3: the best gt bbox is: 2

mapping_from_pred_to_gt: {0: (0, 0.51), 1: (3, 0.81), 2: (1, 0.34), 3: (2, 0.21)}

for i=0, the predicted label: house the ground_truth label: house
for i=1, the predicted label: house the ground_truth label: house
for i=2, the predicted label: watertower the ground_truth label: watertower
for i=3, the predicted label: Dr_Eval the ground_truth label: Dr_Eval




## (b) At iteration 400

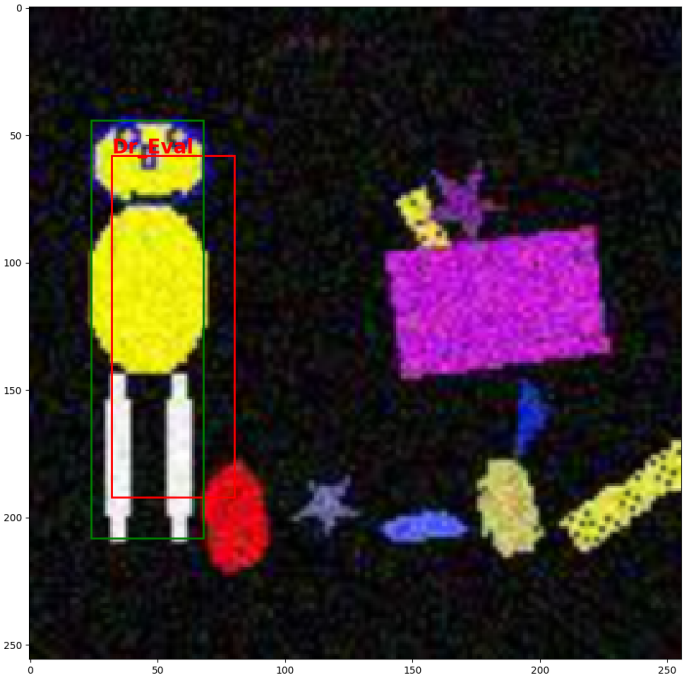
Showing output for test image 400:

For predicted bbox 0: the best gt bbox is: 0

mapping_from_pred_to_gt: {0: (0, 0.32 }

for i=0, the predicted label: Dr_Eval the ground_truth label: Dr_Eval

# Evaluation Results on Unseen Images (contd.)



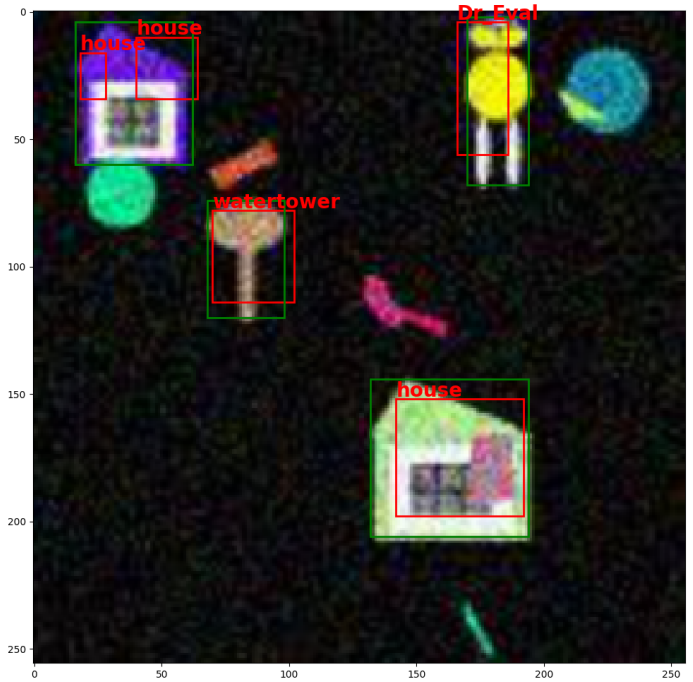

## (a) At iteration 500

Showing output for test image 500:

For predicted bbox 0: the best gt bbox is: 0

mapping_from_pred_to_gt: {0: (0, 0.39)}

for i=0, the predicted label: Dr_Eval the ground_truth label: Dr_Eval




## (b) At iteration 600

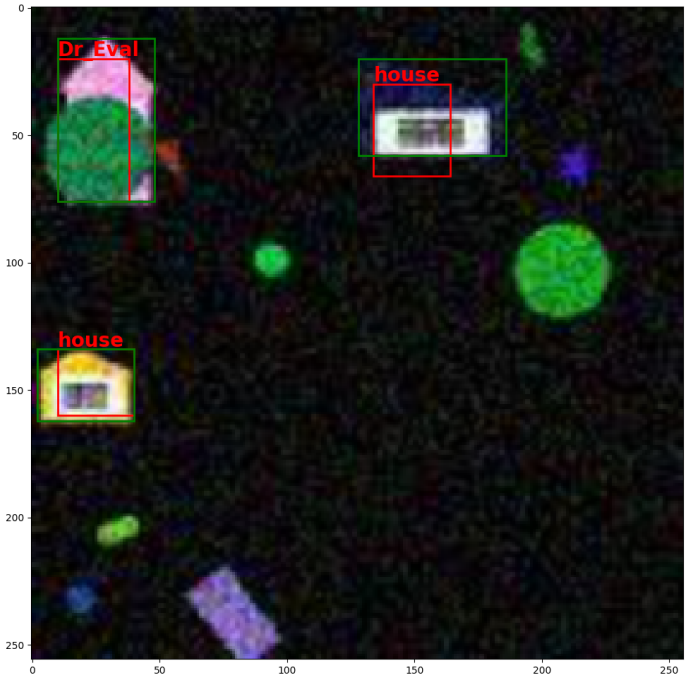
Showing output for test image 600:

For predicted bbox 0: the best gt bbox is: 0
For predicted bbox 1: the best gt bbox is: 2
For predicted bbox 2: the best gt bbox is: 3
For predicted bbox 3: the best gt bbox is: 1
For predicted bbox 4: the best gt bbox is: 3

mapping_from_pred_to_gt: {0: (0, 0.59), 1: (2, 0.44), 2: (3, 0.20), 3: (1, 0.
4: (3, 0.25)}

for i=0, the predicted label: house the ground_truth label: house
for i=1, the predicted label: Dr_Eval the ground_truth label: Dr_Eval
for i=2, the predicted label: house the ground_truth label: house
for i=3, the predicted label: watertower the ground_truth label: watertowe
for i=4, the predicted label: house the ground_truth label: house

# Evaluation Results on Unseen Images (contd.)



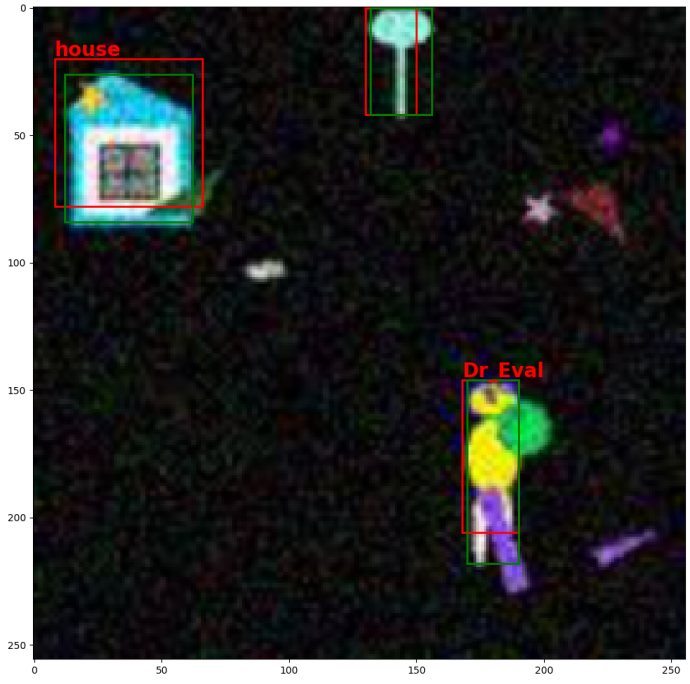

(a) At iteration 700

Showing output for test image 700:

For predicted bbox 0: the best gt bbox is: 1
For predicted bbox 1: the best gt bbox is: 2
For predicted bbox 2: the best gt bbox is: 0

mapping_from_pred_to_gt: {0: (1, 0.75), 1: (2, 0.16), 2: (0, 0.19)}

for i=0, the predicted label: house the ground_truth label: house
for i=1, the predicted label: house the ground_truth label: house
for i=2, the predicted label: watertower the ground_truth label: house




(b) At iteration 800

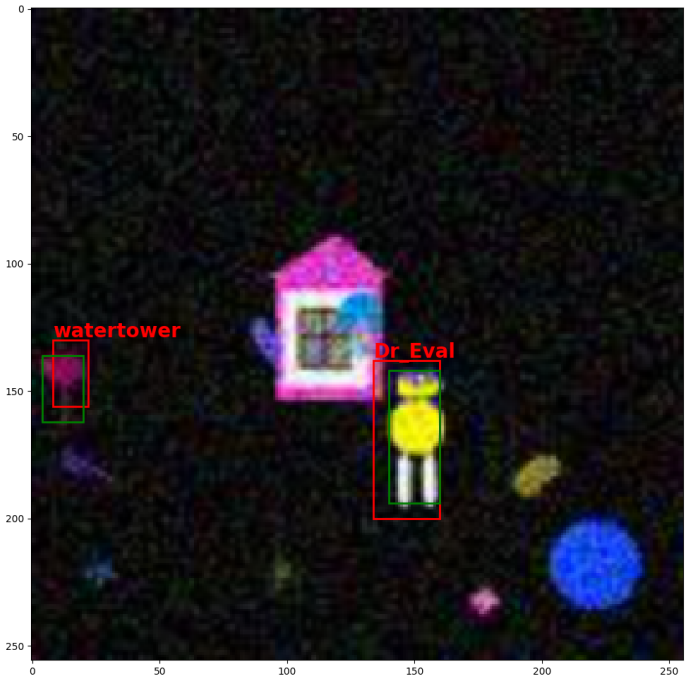
Showing output for test image 800:

For predicted bbox 0: the best gt bbox is: 0
For predicted bbox 1: the best gt bbox is: 2
For predicted bbox 2: the best gt bbox is: 1

mapping_from_pred_to_gt: {0: (0, 0.71), 1: (2, 0.67), 2: (1, 0.58)}

for i=0, the predicted label: house the ground_truth label: house
for i=1, the predicted label: house the ground_truth label: house
for i=2, the predicted label: Dr_Eval the ground_truth label: Dr_Eval

# Evaluation Results on Unseen Images (contd.)



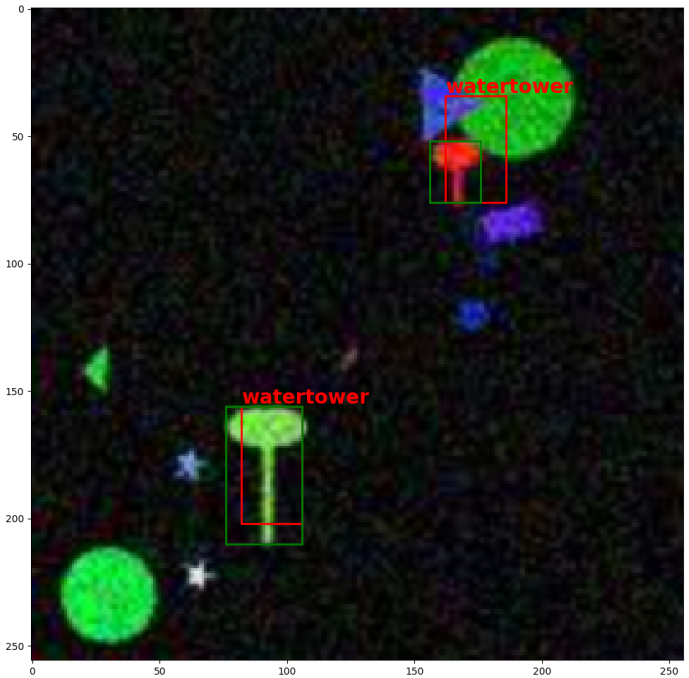

(a) At iteration 900

Showing output for test image 900:

For predicted bbox 0: the best gt bbox is: 0
For predicted bbox 1: the best gt bbox is: 1

mapping_from_pred_to_gt: {0: (0, 0.48), 1: (1, 0.33)}

for i=0, the predicted label: Dr_Eval the ground_truth label: Dr_Eval
for i=1, the predicted label: watertower the ground_truth label: watertower




(b) At iteration 1000

Showing output for test image 1000:

For predicted bbox 0: the best gt bbox is: 0
For predicted bbox 1: the best gt bbox is: 1

mapping_from_pred_to_gt: {0: (0, 0.30), 1: (1, 0.61)}

for i=0, the predicted label: watertower the ground_truth label: watertowe
for i=1, the predicted label: watertower the ground_truth label: watertowe

# Overall Results on the Unseen Test Dataset (After 20 Epochs of Training)

```
Batch size :   4
Learning Rate: 1e-5

Overall accuracy of multi-instance detection on 1000 test images: 88 %          [No hyperparameter tuning]

NOTE 1: This accuracy does not factor in the missed detections. This number is related
        to just the mis-labeling errors for the detected instances.  Percentage of
        the missed detections are presented in the last column of the table shown below.

Displaying the confusion matrix:

                    Dr_Eval          house      watertower         missing

   Dr_Eval:           88.00           0.00           8.00           4.00
     house:            0.00          94.44           4.55           5.56
watertower:            9.09           9.09          45.45          36.36


NOTE 2: 'missing' means that an object instance of that label was NOT extracted from the image.
NOTE 3: 'prediction accurary' means the labeling accuracy for the extracted objects.
NOTE 4: True labels are in the left-most column and the predicted labels at the top of the table.

IoU scores for the different types of objects:

      Dr_Eval:    0.5902
        house:    0.4525
   watertower:    0.4829
```