ECE 60146 Deep Learning Homework 6

Mehmet Berk Sahin, sahinm@purdue.edu, 34740048

March 22, 2023

1 Introduction

In the previous homework, we worked on an object detection task and we assumed each image includes a single object, which is supposed to be detected by using residual blocks. In addition to that task, in this lab, we are interested in multi object detection where an image can include multiple objects, which need to be detected and classified. You Only Look Once (YOLO) architecture is an efficient network, which was designed for multi-object detection, and it is so efficient that it can even make predictions in real-time [1]. The purpose of this lab is **i)** understanding the logic behind YOLO, in other words, how multi-instance detection is achieve via single forward pass, and **ii)** implementing our own YOLO training and evluation logic. Since YOLO logic is used both in traininig and inference phases, I want to first explain it to refer it in subsequent sections.

1.1 YOLO Logic

Briefly, YOLO logic assigns cells and anchor boxes to objects in an image to increase the efficiency of its localization and make the inference and training over one network. Doing so, it achieves real-time classification performance. Every image in the dataset is divided into grid of cells and for each cell, a set of anchor boxes having different aspect ratios are defined. Cells have square shapes and length of their edge is called `yolo_interval`. And aspect ratio of anchor boxes are defined as the ratio between its width and height. Instead of looking at the whole image as we did in the previous assignment, the model needs the look at the cell number and anchor index pairs to draw the bounding box and make classification. Each object corresponds to one `(cell, anchor_box)` pair and it is represented by `yolo_vector`. Its structure is given below:

`(obj,` δx`,` δy`, h, w, obj_class)`

`obj` is 1 if there is an object in the cell-anchor box pair otherwise it is 0. δx, δy terms denote the distance between the center of the cell and the corresponding bounding box center. `h` and `w` represent height and width of the bounding box divided by the height and width of the cell. Lastly, the last term is the one-hot-encoded representation of the object class in the bounding box. These `yolo_vectors` are elements of `yolo_tensor`, whose shape is given below:

`(batch_size, num_cells, num_anchors, 8)`

First axis represents the number of images in one batch. Second axis denotes number of cells in the grid over an image. For example, if `yolo_interval`$= 20$ and image size is 256×256, then the grid will be 12×12 and there will be 144 cells per image. Note that 16 pixels are not included to the grid and are completely ignored due to YOLO logic. Third axis denotes the number of anchor boxes. In the examples that we covered in the class, aspect ratios of anchor boxes are as follow:

`1/5, 1/3, 1 3/1, 5/1`. So, aspect ratio of the ground truth bounding box is calculated and its `yolo_vector` is saved into the corresponding anchor box or the one that has the closest aspect ratio index in `yolo_tensor`. Lastly, the fourth axis is allocated to `yolo_vector`. The idea here is dividing the problem into subproblems and doing the training and inference in a single network with real-time performance. YOLO was not the best model in the classification at the time paper was published but it was the best model in real-time object detection.

Rest of the paper is organized as follows. In Section 2, I explained pre-processing on COCO dataset and structure of it with the explanation of the code. In Section 3, I explained the train and inference logic for YOLO multi-object detection. I explained the relevant codes. In Section 4, I presented the learning curves and example classifications, and I discussed them. In Section 5, I discussed the lessons learned from this homework.

2 Dataset Preparation

Data preparation is similar to the previous homeworks. Train and test images come from **2014 Train images** and **2014 Test images** folders in COCO webpage. And their annotations are in **2014 Train/Val annotations**. Data hierarchy is as follows: folders are located under **coco** folder and their names are **train2014**, **test2014**, and **annotations2014**. Before running the program, you should input the path of the parent folder of **coco** to the program. I will explain how to run the code later. To create the dataset that is used in the training of YOLO, I implemented `YoloDataset` class in `dataset.py` module.

Data generation part is very similar to the previous homework except that I implemented `yolo_extractor()` function. In homework 5, for each image in COCO dataset, I resized the image and its bounding box, then saved to the local disk with its label. Different from this, I saved each image file with the information of the objects' cell indices, anchor indices, labels, and `yolo_tensor` of the whole image. Code snippet for the call of `yolo_extractor` is given in Figure 1. As most of the part is similar to my implementation in HW5 and similar to the code in HW5, I do not explain `data_generator()` function again. As you may notice, the part that comes before the call

```
# load the image with resizing
img = self.coco.loadImgs(img_id)[0]
I = io.imread(os.path.join(self.data_path, img['file_name']))
if len(I.shape) == 2:
    I = skimage.color.gray2rgb(I)
img_h, img_w = I.shape[0], I.shape[1]
I = resize(I, (self.img_size, self.img_size), anti_aliasing=True, preserve_range=True)
image = np.uint8(I)
# scale annotation bounding boxes
x_scale, y_scale = self.img_size / img_w, self.img_size / img_h
# get yolo_tensor and other annotations
cell_anc_cat, yolo_tensor = self.yolo_extractor(anns, x_scale, y_scale)
data[img['file_name']] = {"cell_idx" : cell_anc_cat[:,:2],
                          "anchor_idx" : cell_anc_cat[:,2],
                          "label" : cell_anc_cat[:,3],
                          "yolo_tensor" : yolo_tensor}
# save the image
if self.train:
    io.imsave(os.path.join("train_data", img['file_name']), image)
else:
    io.imsave(os.path.join("test_data", img['file_name']), image)
```

Figure 1: Call of `yolo_extractor` under `data_generator()`

of `yolo_extractor` is very similar to HW5. Different from that, I passed the annotations and

scales of x, y as arguments to the `yolo_extractor` function. As an output, I get cell ids, anchor box ids and labels of the objects in the image and image's `yolo_tensor` representation. I saved these objects into my "data" dictionary object whose keys are file names of the images and whose values are dictionary of cell ids, anchor ids, labels and `yolo_tensor` of images. Thus, my dataset is structured as dictionary and saved to the local disk to be used later in the training. And resized images are saved in to the local folders which are "train_data" and "test_data". Next, I will explain the `yolo_extractor()`.

2.1 yolo_extractor()

This function is an instance function of `YoloDataset` class in `dataset.py` module. I will explain how `yolo_vector` and `yolo_tensor` are generated from bounding boxes in COCO. First part of `yolo_extractor()` is given in Figure 2. It takes annotation list (`anns`) and scales for x, y coordi-

```python
def yolo_extractor(self, anns, x_scale, y_scale):
    yolo_tensor = np.zeros((self.num_cells_width * self.num_cells_height, self.anchor_num, 8))
    # save cell index, anchor num and label
    cell_anc_cat = np.zeros((self.max_obj, 4))
    cell_anc_cat[:,-2:] = 13 # 13 means there is no object
    for i, ann in enumerate(anns):
        # take the bounding box
        class_idx = self.coco_inv_labels[ann['category_id']]
        [x, y, w, h] = ann['bbox']
        # scale the images due to resizing
        [x, y, w, h] = [x * x_scale, y * y_scale, w * x_scale, h * y_scale]
        # bbox center
        x_center, y_center = y + h/2, x + w/2
        # cell index (i, j)
        row_cell_idx = min(x_center // self.yolo_interval, self.num_cells_height - 1) # ith row
        col_cell_idx = min(y_center // self.yolo_interval, self.num_cells_width - 1) # jth column
        # bounding box scale
        bw = w / self.yolo_interval
        bh = h / self.yolo_interval
```

Figure 2: `yolo_extractor()` implementation part 1

nates due to resizing of images. Note that the shape of `yolo_tensor` is `(num_cells, num_anchors, 8)` and the shape of `cell_anc_cat` is `(max_num_objects, 4)`, where `max_num_objects` mean maximum number of objects in an image. This number equals to the maximum number of objects in an image over the training dataset. Then, for each annotation or bounding box, I perform following operations: **i)** convert the COCO category index to my index convention `(bus=0, cat=1, pizza=2)`. **ii)** using scales for x and y, calculate the center of the bounding box. **iii)** using the center of the bounding box, maximum row and column cell numbers, and cell width, which is `yolo_interval`, calculate the row and column cell index that corresponds the bounding box and calculate its height and width as multiple of `yolo_interval`. Then, **iv)** calculate cell centers that are the closest to

```python
        cell_i_center = row_cell_idx * self.yolo_interval + self.yolo_interval / 2
        cell_j_center = col_cell_idx * self.yolo_interval + self.yolo_interval / 2
        # calculate difference between centers
        dx = (x_center - cell_i_center) / self.yolo_interval
        dy = (y_center - cell_j_center) / self.yolo_interval
        # aspect ratio
        AR = h / w
        if AR <= 0.2:               anc_idx = 0
        if 0.2 < AR <= 0.5:         anc_idx = 1
        if 0.5 < AR <= 1.5:         anc_idx = 2
        if 1.5 < AR <= 4.0:         anc_idx = 3
        if 4.0 < AR:                anc_idx = 4
        yolo_vector = np.array([1, dx, dy, bh, bw, 0, 0, 0])
        yolo_vector[5 + class_idx] = 1
        # save the yolo_vector to yolo_tensor
        yolo_tensor[int(row_cell_idx * self.num_cells_width + col_cell_idx), anc_idx, :] = yolo_vector
        cell_anc_cat[i,:2] = (row_cell_idx, col_cell_idx)
        cell_anc_cat[i, 2] = anc_idx
        cell_anc_cat[i, 3] = class_idx

    return cell_anc_cat, yolo_tensor
```

Figure 3: `yolo_extractor()` implementation part 2

the bounding box by using cell dimension and cell indices. **v)** calculate the difference between the

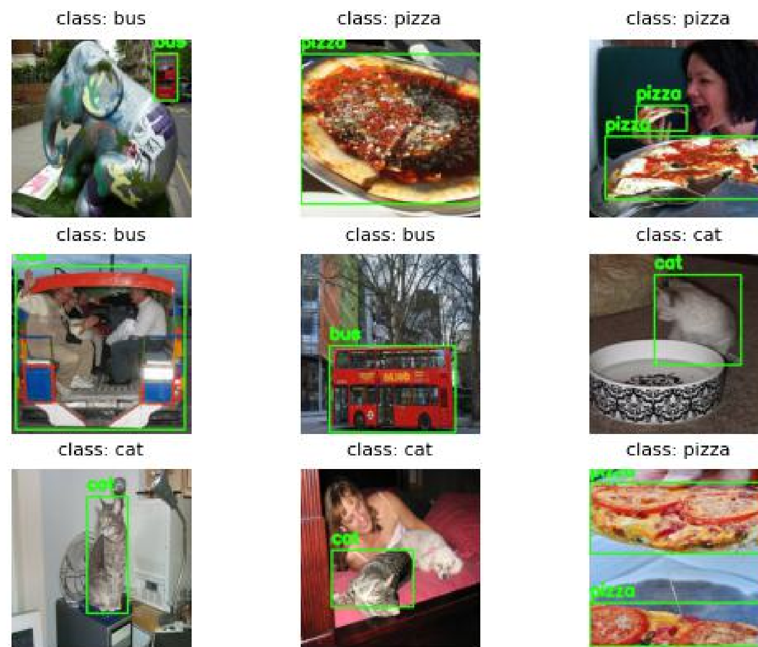


Figure 4: Randomly chosen samples from training set

bounding box and the cell center (δx and δy) as a multiple of cell width by taking the difference of them and dividing them by `yolo_interval`. Until now, δx, δy, `h`, and `w` of `yolo_vectors` are calculated. Row and column cell indices are found for each bounding box. However, we did not assign any anchor box to ground truth bounding boxes as suggested in homework. Instead of defining hand-crafted anchor boxes, I defined intervals for aspect ratio as we discussed in class because I think the model's performance will depend on my choice of anchor box less than the former method. I defined 5 anchor boxes whose aspect ratio are $1/5, 1/3, 1, 3/1$, and $5/1$. Then, **vi)** I calculated the aspect ratio of each bounding box and assign anchor box index whose aspect ratio is the closest to the bounding box to that object as can be seen in Figure 3. Then, **vii)** I saved the category of the object in one-hot-encoded representation to the last 3 elements of the **yolo_vector**. Lastly, **viii)** I saved `obj=1`, δx, δy, `h`, `w` and one-hot-encoded `obj_class` to `yolo_vector`. And function returns two tensors: first one consists of cell and anchor box indices with labels, and the second one is the `yolo_tensor`.

2.2 `data_generator()`

Implementation of this function is very similar to the previous homework so I will not explain it again but I will give the rules that I follow to generate the dataset. They are as follows:

- Each image contains at least one *foreground* object.
- Each *foreground* object must belong to one of the three categories: `['bus', 'cat', 'pizza']`.
- The area of any *foreground* object must exceed $64 \times 64 = 4096$ pixels.

- If there is no *foreground* object in an image, do not include the image to the dataset.
- Resize each image to 256×256 and scale the bounding boxes accordingly.

I randomly sampled images from both train and test dataset as asked in the assignment. Train samples can be sen on Figure 4. And test samples can be seen on Figure 5. I provided 3 images from

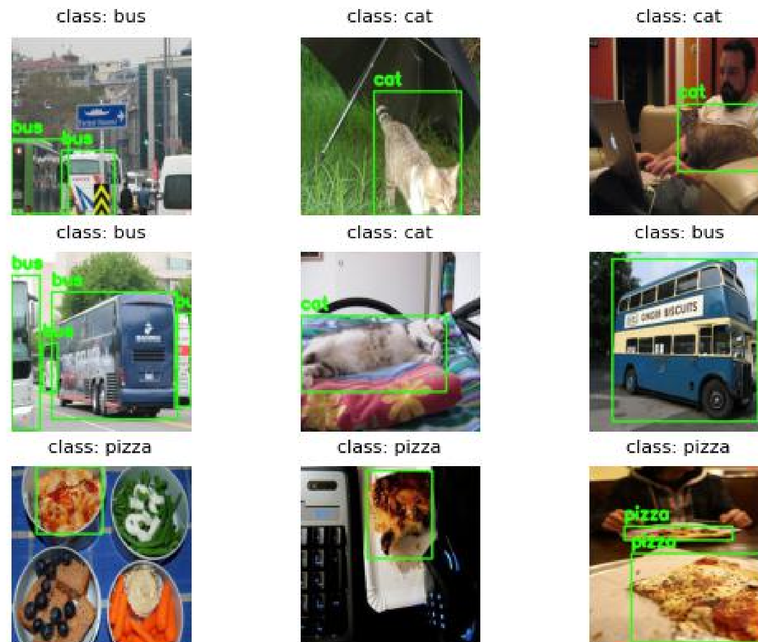


Figure 5: Randomly chosen samples from test set

each class for both training and test sets as asked in the assignment. As you may notice, objects in the images are not always obvious. For example, in training set, in image at row 1 and column 3, although there is a very large pizza, there is small pizza slice, which is about to be eaten by woman. And it is not flat, it has some curvature. Due to these facts, learning its bounding box and class may be difficult for the network. Another example in test set, at row 3 and column 3, there are 4 meals one of which is pizza. Although other meals do not belong to pizza class, they consists of some texture that is common in foods. So, that may increase the difficulty of localization. Lastly, in test set, at row 2 column 3, tail of the bus behind the large bus appears and it constitutes very small bounding box. And at the left there is less than half view of a buss. That kind of incomplete objects make localization and classification harder.

Lastly, I checked the sizes of my train and test sets after filtering according to the criteria mentioned above. In the **training set**, I obtained **6883** images and in the **test set**, I obtained **3491** images. The former is above **6000** and the latter **3000**, which show that I satisfied the criteria in homework. Next I will explain the training and inference logic with the code.

3 YOLO Train & Inference Logic

In this section, I will first explain the training and inference logic of my YOLO model, then I will explain the important parts of the code. For the full source code, you can check the attachments. I think they are well-commented. I will not explain my network architecture because I used the same network that I used in previous homework. Its CNN backbone is similar to the recommended architecture in HW5 and it uses `BasicBlock` which are skip-connections and which is used in `Resnet34` [2]. At the last layer, I implemented single layer fully connected layer in order to obtain a `yolo_tensor` as the output of the network. I tried to implement more than one fully connected layer but number of parameters increased drastically and convergence slowed down so I tuned it to single layer.

3.1 YOLO Train Logic

In preparation of the dataset, each object in an image is represented by `yolo_vector`. Each `yolo_vector` consists of objectness bit, row and column cell indices, anchor id, and class of the object. So, model needs to decide whether there is an object for each grid cell and anchor box pairs. Since this is a binary classification task, its loss should be Binary Cross Entropy Loss (BCE Loss). If the objectness bit is 1, then model needs to draw a bounding box. It does this by doing regression (nonlinear) on ground truth values of center offsets, and width and height of the ground truth bounding box. As the method is regression, its loss is chosen as Mean Squared Error (MSE). This loss is calculated only for cell-anchor pairs whose objectness bit is 1. In addition to the location of the predicted bounding box, the model should also predict the class of the object, which can be bus, cat or pizza in our setting. The model performs this by predicting the last 3 number of the `yolo_vector` correctly. For example, if the object in row cell 1, column cell 2 and anchor box 3 is cat, which is encoded as $(0, 1, 0)$, then predicted `yolo_vector` corresponding to the same cell-anchor pair location should have the highest value at 6th index. Ideally, that entry should be infinity and 5th and 7th entries should be minus infinity. Since the task is multi-classification, I used Cross Entropy Loss (CE Loss). I combined these under one loss function given below:

$$
\begin{aligned}
Loss =& \lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{i,j}^{obj}\left[(\delta x_i-\delta\hat{x}_i)^2+(\delta y_i-\delta\hat{y}_i)^2\right]\\
&+\lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{i,j}^{obj}\left[(w_i-\hat{w_i})^2+(h_i-\hat{h}_i)^2\right]\\
&+\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{i,j}^{obj}\left[p(obj_{i,j})\log(p(o\hat{b}j_{i,j}))+(1-p(obj_{i,j}))\log(1-p(o\hat{b}j_{i,j}))\right]\\
&+\lambda_{noobj}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{i,j}^{noobj}\left[p(obj_{i,j})\log(p(o\hat{b}j_{i,j}))+(1-p(obj_{i,j}))\log(1-p(o\hat{b}j_{i,j}))\right]\\
&+\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{i,j}^{obj}\left[\sum_{c\in classes}p(c_{i,j})\log(p(\hat{c}_{i,j}))\right],
\end{aligned}
\tag{1}
$$

where $p(obj_{i,j})$ is the ground truth for objectness, $p(o\hat{b}j_{i,j})$ is predicted probability of objectness, $p(c_{i,j})$ is ground truth for class c (either 1 or 0), and $p(\hat{c}_{i,j})$ is the predicted probability of class c for each ith cell and j^{th} anchor pair. This loss is inspired by the loss proposed in the original

YOLO paper [1]. It is similar but not not exactly the same. As you may noticed, there are two weighting coefficients: λ_{coord} and λ_{noobj}. The first one is for compensating the low number of `yolo_vectors` that include objects. For example, if cell widths is 20, image size is 256, and number of anchor box is 5, and if there is only one object in the image, then other 719 `yolo_vector` are not included into MSE loss as they do not include any object. As a result of this, gradients for correcting bounding boxes will be very small compared to other tasks' gradients. Thus, I decided to increase its importance by setting $\lambda_{coord} > 1$. Similarly, as there are lot of ground truths with objectness=0, I scaled the corresponding BCE loss with $0 < \lambda_{noobj} < 1$. In the original YOLO paper, objects are assigned to cells [1], in my implementation, different than that, I assigned them to cell-anchor box pairs. So, CE over classes are summed over anchor boxes as well. Lastly, in the original paper, authors performed regression [1], but I used BCE and CE losses for classification tasks. I conducted the calculation of this loss function without any for loop over images, cells, and anchor boxes.

3.2 YOLO Train Code & `run_train()`

Training code of the YOLO is implemented in `model.py` module. It is an instance function of `YoloNet` so `self` in the code represents the YOLO model. I divided the code into parts to explain it step-by-step. I will go through the important steps and ignore the common things like loading the model to device, which we do in every homework. For details, one can check the source code. First part can be seen on Figure 6. It takes train and test datasets as arguments to train the model

```python
def run_train(self, train_loader, test_loader, epochs=50, lr=1e-3, betas=(0.9, 0.99)):
    # load model to device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    self = self.to(device)
    # losses
    criterion1 = nn.BCELoss()
    criterion2 = nn.MSELoss()
    criterion3 = nn.CrossEntropyLoss()
    # choose optimization method
    optimizer = optim.Adam(self.parameters(), lr, betas)
    start_time = time.perf_counter()
    train_loss_hist = {"bce" : [], "mse" : [], "ce" : []}
    test_loss_hist = {"bce" : [], "mse" : [], "ce" : []}
    min_loss = 100000000000
    print("Training is running...")
```

Figure 6: YOLO training code part 1

and report the train and test performance at each epoch. And betas are for Adam optimizer. Using PyTorch's `torch.nn` module, I initialized the BCE, MSE and CE loss objects and I initialized the dictionaries to save the train and test losses to be displayed later. The next part can be seen in Figure 7. The outer loop is the main training loop, which exists in almost every deep learning code, and the inner loop is for iterating over the train dataset with mini-batches. Other lines are initialization of the variables to be used in the training. `loss` corresponds to *Loss* in equation (1). `objectness` keeps 1s and 0s for each image's cell and anchor pair. And note that `yolo_tensor` consists of ground truth and its shape is `(batch_size, num_cells, num_anchors, 8 )`. `output` is vector and prediction of the model. It is reshaped to the shape of `yolo_tensor`. Lastly, `yolo_idx` is the indices of images, cells and anchor boxes triples the include objects. After initializations, we can calculate the BCE and MSE losses as given in Figure 8. I calculate BCE for object and no object cases. `tmp` consists of indices with `objectness`=0 and `yolo_idx` consists of indices with `objectness`=1. Using these, I filtered ground truths and corresponding predictions. I passed the predictions through sigmoid function and calculated the BCE loss. To calculate the MSE loss, I reshaped the `yolo_tensor` and `output` to the following form:

`(batch_size, num_row_cell, num_col_cell, num_anchors, 8)`

```python
# training loop
for epoch in range(epochs):
    # losses that will be reported at each epoch
    run_loss_bce = 0.0
    run_loss_ce = 0.0
    run_loss_mse = 0.0
    self.train()
    for data in train_loader:
        # extract data
        imgs, gts = data
        yolo_tensor, label = gts['yolo_tensor'], gts['label']
        # load them to gpu or cpu
        imgs = imgs.to(device)
        yolo_tensor = yolo_tensor.float().to(device)
        # forward pass
        optimizer.zero_grad()
        output = self(imgs)
        output = output.view(yolo_tensor.shape)

        # CALCULATE TOTAL LOSS
        loss = torch.tensor(0.0, requires_grad=True).float().to(device)
        objectness = yolo_tensor[:,:,:,0] # objectness ground truths
        yolo_idx = torch.nonzero(objectness) # consider cell&anch that have objects
```

Figure 7: YOLO training code part 2

```python
# CALCULATE BCE LOSS
# calculate binary cross entropy for no object cases
tmp = torch.nonzero(objectness == 0)
no_obj_pred = output[tmp[:,0], tmp[:,1], tmp[:,2]]
no_obj_gt = yolo_tensor[tmp[:,0], tmp[:,1], tmp[:,2]]
loss_no_obj = criterion1(nn.Sigmoid()(no_obj_pred[:,0]), no_obj_gt[:,0])
# calculate binary cross entropy for objects cases
obj_pred = output[yolo_idx[:,0], yolo_idx[:,1], yolo_idx[:,2]]
obj_gt = yolo_tensor[yolo_idx[:,0], yolo_idx[:,1], yolo_idx[:,2]]
loss_obj = criterion1(nn.Sigmoid()(obj_pred[:,0]), obj_gt[:,0])
# save the loss for bce
run_loss_bce += loss_no_obj.item() + loss_obj.item()
# CALCULATE MSE LOSS
yolo_tensor = yolo_tensor.view(-1, self.max_row_cell, self.max_col_cell, self.anchor_num, 8)
output = output.view(yolo_tensor.shape)
# pull cell numbers of objects
objects = torch.nonzero(label != 13) # only include real objects
cell_nos = gts['cell_idx'][objects[:,0], objects[:,1]].type(torch.long)
anchor_nos = gts['anchor_idx'][objects[:,0], objects[:,1]].type(torch.long)
#anchor_nos = gts['anchor_idx'][objects[:,0], objects[:,1]]
output = output[objects[:,0], cell_nos[:,0], cell_nos[:,1], anchor_nos]
yolo_tensor = yolo_tensor[objects[:,0], cell_nos[:,0], cell_nos[:,1], anchor_nos]
# calculate mse loss
mse_loss_obj = criterion2(output[:,1:5], yolo_tensor[:,1:5])
run_loss_mse += mse_loss_obj.item()
# calculate multi class cross entropy loss
labels = torch.argmax(yolo_tensor[:,5:], dim=1)
tmp = criterion3(output[:,5:], labels)
run_loss_ce += tmp.item()
```

Figure 8: YOLO training code part 3

Then, using the ground truths for cell numbers and anchor box ids, which come from `gts` dictionary, I filtered the `yolo_vectors` that correspond to the objects in images from mini-batch. This filtering is done above "calculate mse loss" comment. Then, using `nn.MSELoss()` object, I calculated the MSE loss for δx, δy, `h`, and `w`. And I multiplied the loss by λ_{coord}. Doing so, I calculate the first two row of equation (1). And BCE calculation explained previously correspond to third and fourth row of equation 1. Lastly, I calculated the cross entropy loss for multi-classification only for predictions whose corresponding ground truth's objectness is 1. It can be seen on the last 3 lines in Figure 8. After those lines I performed backward pass on `loss` and take a step on `optimizer`.

After the inner loop is completed, I save and report the running losses, which are BCE, MSE, and CE losses per epoch. Then, I take the model to evalution mode with "`eval()`" function of PyTorch. And very similar inner loop for the test set was conducted except that there is not any gradient updates. I only evaluated the performance of the model over the test set to see whether there is any overfitting. Details can be found in the source code.

3.3 YOLO Inference Logic

After the training, model needs to predict bounding boxes for the objects in an image. Hopefully, this prediction should belong to the correct class. In this section, I will explain my inference YOLO logic and go through the important parts of the code.

In the training, loss was calculated given the knowledge of ground truths. As opposed to that, in the inference, this is impossible as we do not have an access to the labels or cell-anchor box pairs that include objects. Thus, we cannot choose its corresponding predicted `yolo_vector` and report this as a predicted bounding box. However, setting a threshold for objectness, one can report the predicted bounding boxes. If algorithm generalizes well to the data and if the threshold is set properly, then predicted objects that exceeds the threshold should be in the same cell and anchor box ids with its ground truth and its classification should match with it. Although this approach seems good, it has a shortcoming. In prediction `yolo_tensor`, there can be several bounding boxes with high confidence in `objectness`. To overcome this, I used Non-maximum Supression algorithm in the inference.

3.4 Non-maximum supression (NMS)

This algorithm removes the additional bounding boxes for the same object and make the model display the most confident bounding box per object in an image. It is as follows: `yolo_vector`s are divided into groups with respect to their predicted classes. For each group, `yolo_vector` with the highest confidence is chosen. In our setting, it is the predicted probability of the objectness. Then, intersection over union (IoU) scores were calculated between the most confident bounding box and other bounding boxes for each class. Bounding boxes whose IoU score exceed the IoU threshold are removed from the list for each class. And remaining boxes are displayed as final predictions. This algorithm assumes that if there are multiple objects with the same class in an image, they should be far from each other because if they are close, then their IoU should be high and one of them which has a lower confidence will be eliminated. However, it mitigates the problem of multiple bounding boxes to some extent.

As opposed to previous homeworks, performance of the model will be evaluated quantitatively rather than qualitatively. Because examining the performance of object detector is much more complicated than evaluating a classifier or regressor and it can be beyond the scope of this HW. So, I will present the train and test results and comment on them.

3.5 YOLO Test Code & `inference()`

Inference/Test code of the YOLO was implemented in `model.py` module. It is an instance function of `YoloNet` so `self` in the code represents the YOLO model. I will go through the important steps and you can check the source code for more.

The first part of `inference()` can be found in Figure 9. It has 3 input variables: `model_path`, `test_loader`, and `sample_num`. The first one is the path for a model whose results will be displayed. The second variable is `Dataloader` which consists of either train or test samples. The last variable is the number of samples to be displayed per row and there are three row in the figure. So, if `sample_num=2`, then in total, there should be 6 images with predictions and ground truth bounding boxes. First lines in Figure 9 consists of loading model from local disk, initalize the

```python
def inference(self, model_path, test_loader, sample_num=2):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # set device
    class_list = ['bus', 'cat', 'pizza'] # class list
    # load model
    self.load_state_dict(torch.load(model_path))
    self.to(device)
    self.eval()
    # set the figure and axes for display
    fig, axs = plt.subplots(3, sample_num)
    fig.tight_layout()
    # iterate through data loader, batch size = 1
    for k, data in enumerate(test_loader):
        # display only sample_num number of images per
        if k > 3 * sample_num - 1:
            break
        # extract the annotations and images
        imgs, gts = data
        yolo_tensor, anchor_idx = gts['yolo_tensor'], gts['anchor_idx']
        cell_idx, labels = gts['cell_idx'], gts['label']

        # prepare image for display
        img = np.uint8(imgs[0].numpy() * 255)
        img = img.transpose((1, 2, 0))
        img = np.ascontiguousarray(img)

        # load them to gpu or cpu
        imgs = imgs.to(device)
        yolo_tensor = yolo_tensor.float().to(device)
        output = self(imgs)
        # reshape the prediction and labels to (num. row cell, num. col cell, anchor id, 8)
        output = output.view(yolo_tensor.shape)
        yolo_tensor = yolo_tensor.view(self.max_row_cell, self.max_col_cell, self.anchor_num, 8)
```

Figure 9: `inference()` implementation part 1

figure object from `matplotlib.pyplot` package and setting the device and class indices. For loop iterates through the given dataset with **batch size 1**, which was determined before this function call. If the iteration number exceed the total number to be plotted, then program quits for loop. After that if statement, I extracted the ground truths, put the image tensor into displayable image format and conducted forward pass via "`self(imgs)`" line. Lastly, ground truth `yolo_tensor` and `output` of the network is reshaped into the structure in Section 3.2. The second part of the function can be seen in Figure 10. Before explaining it, I need to mention the convention I followed

```python
# forward pass for inference
yolo_idx = torch.nonzero(anchor_idx[0] != 13)[0]
# iterate through objects to display ground truths
for obj_idx in range(len(yolo_idx)):
    # extract ground truths for each object
    label = int(labels[0, obj_idx].item())
    row_cell_idx, col_cell_idx = cell_idx[0, obj_idx]
    anch_idx = anchor_idx[0, obj_idx]
    [row_cell_idx, col_cell_idx, anch_idx] = list(map(lambda x: int(x.item()), [row_cell_idx, col_cell_idx, anch_idx]))
    # relevant yolo vector
    yolo_vector = yolo_tensor[row_cell_idx, col_cell_idx, anch_idx]
    # calculate ground truth bbox size
    h = yolo_vector[3].item() * self.yolo_interval
    w = yolo_vector[4].item() * self.yolo_interval
    # calculate cell centers
    cell_i_center = row_cell_idx * self.yolo_interval + self.yolo_interval / 2
    cell_j_center = col_cell_idx * self.yolo_interval + self.yolo_interval / 2
    # calculate the center of gt bbox
    x_center = yolo_vector[1].item() * self.yolo_interval + cell_i_center
    y_center = yolo_vector[2].item() * self.yolo_interval + cell_j_center
    [x1, y1, x2, y2] = [round(y_center-w/2), round(x_center-h/2), round(y_center+w/2), round(x_center+h/2)]
    # draw the bounding box
    img = cv2.rectangle(img, (round(x1), round(y1)),
                        (round(x2), round(y2)),
                        (36, 256, 12), 2)
    # draw its class name
    img = cv2.putText(img, class_list[label], (round(x1), round(y1 - 10)), cv2.FONT_HERSHEY_SIMPLEX,
                      0.8, (36, 256, 12), 2)
```

Figure 10: `inference()` implementation part 2

in `data_generator()` to prepare my dataset. As mentioned previously, there can be at most 14 objects in one image. So, for the image in which there are less than 14 objects, remaining entries were filled with 13 meaning that there is no object at that entry. Therefore, in the first line of part 2, I filtered the anchor boxes that are not equal to 13 so that I will get the number of objects in an image, which is equivalent to `len(yolo_idx)`. For loop iterates through the objects and following 4 lines extract the anchor box id, row and column cell ids and the class label for the object from

ground truths. To get the coordinates of top left and bottom right of the bounding box, the same logic in Figure 2 and 3 were followed, which was explained in Section 2.1. In the last two lines, `cv2` package was used to draw the ground truth bounding box with its label as we did in last homework.

Up to this point, program draw the ground truth bounding boxes for each object. What remains is drawing the predicted bounding boxes, which is implemented in Figure 11. In the first line, object

```
# objectness prediction
output[0,:,:,0] = nn.Sigmoid()(output[0,:,:,0])
cobj_idx = torch.nonzero(output[0,:,:,0] > self.threshold)
box_cand = output[0, cobj_idx[:,0], cobj_idx[:,1]]
# extract corners of the predicted bounding box
pred_boxes = self.bbox_to_corners(cobj_idx[:,0], box_cand, device)
bboxes = []
# predicted categories
pred_cats = torch.argmax(pred_boxes[:,5:], dim=1)
# perform non-max supression for each class
for i in range(3):
    idxs = torch.nonzero(pred_cats == i)
    if idxs.shape[0] != 0:
        # non-max supression for objectness = 1
        class_tensors = pred_boxes[idxs[:,0]]
        indices = nms(boxes=class_tensors[:,1:5], scores=class_tensors[:,0], iou_threshold=self.iou_ths)
        bboxes.append(class_tensors[indices,:])
```

Figure 11: `inference()` implementation part 3

predictions are passed through Sigmoid activation function to estimate probability of objectness. Then, `yolo_vectors` with objectness not exceeding the threshold are filtered out and `box_cand` only includes `yolo_vectors` that includes object(s) with high probability or confidence. Then, to get the top left and bottom right corners of the predicted bounding boxes, `box_cand` is passed through `bbox_to_corners()` function. It does a very similar job as in Figure 2, 3 ,and 10, which I explained previously. After that, predicted class indices were acquired via `torch.argmax()` function. It is followed by a for loop which iterates through class indices. For each class, it performs NMS and `yolo_vectors` that exceed the IOU threshold are added to a list. Lastly, `yolo_vectors` in `bboxes` are plotted as in the last lines of Figure 10.

4 Results

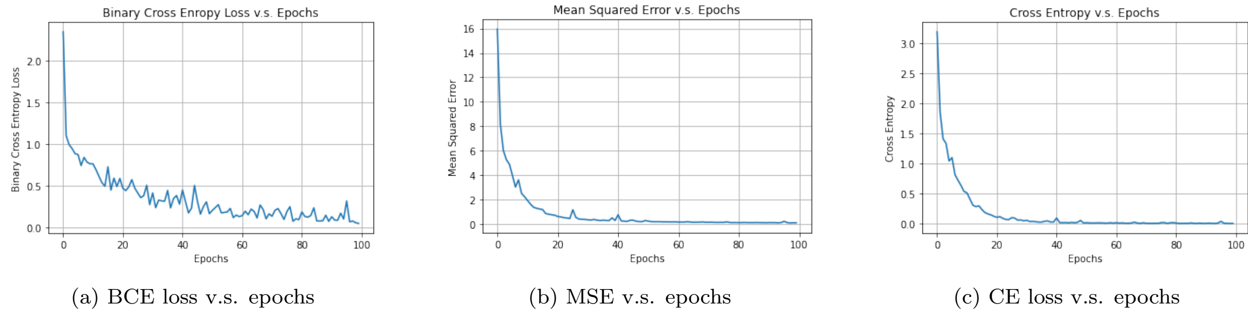
In this section, I will present the results of both training and test sets with the sample predictions from those sets. And to demonstrate the learning of the algorithm, I will provide the reader with the training loss curves for BCE, MSE and CE losses.

4.1 Experiment Setting

I conducted the experiment with the following hyperparamters: learning rate is 10^{-3}, `yolo_interval` is 20, number of anchor boxes is 5, batch size is 32, image size is 256×256, objectness threshold is 0.9, IOU threshold is 0.4, $\lambda_{coord} = 5$ and $\lambda_{noobj} = 0.5$ as in [1], and the training is run over 100 epochs. To prevent overfitting, I saved the best model into memory. And the best model is determined based on the average test performance of the model over BCE, MSE and CE losses. Training was conducted on NVIDIA A100 GPU.

4.2 Training Results

In this section, I will present plots for training losses per epoch to demonstrate that my model is learning and YOLO logic works correctly. Then, I will present sample predictions from training dataset. Training losses over epochs can be found in Figure 12. As it can be seen on the loss plots,



(a) BCE loss v.s. epochs (b) MSE v.s. epochs (c) CE loss v.s. epochs

Figure 12: Training losses per epoch

all losses get very close to 0. That implies power and complexity of my model is sufficient the learn the data so it did not underfit. Losses in the plots are mean values over the training data. Although training is done with weighting coefficients, these losses are saved without any weighting, which are λ_{noobj} and λ_{coord}. And MSE and CE losses are calculated over the predictions that corresponds to the `yolo_vectors` with objectness 1. In Figure 12, plot of MSE and CE is smoother than BCE and BCE is noiser than others. You can see the plot of all losses together in Figure 13. I seems

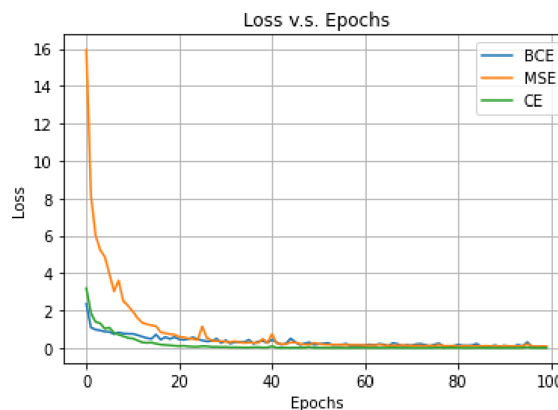


Figure 13: Losses v.s. epochs

that there is more loss in MSE than CE and BCE losses so it decreases more than other losses. Another observation is that no matter where the losses started, all of them converged on 0, which is the global minimum. That indicates model is powerful enough so I did not try to implement more powerful or more complex model to increase the accuracy of the predictions. Additionally, from Figure 12, it seems that model is capable of learning to detect whether there is an object in any region of the image (objectness), the sizes of the bounding box (localization), and determine its class (multi-instance classification).

To demonstrate the performance of the model on training dataset, I picked some samples from training set and plot them on 3×3 grid, which can be seen in Figure 14. Since the task is *multi-instance* object detection, I put images having more tha one objects to see multi-object detection performance of the training model. Ground truths are denoted with green bounding boxes and predictions of my YOLO model are denoted with red bounding boxes. And corresponding class names for bounding boxes can be found on the top left corner of each box. Each row of Figure 14 and 15 is allocated to one class, which can be bus, cat or pizza. Row and column indices of 3×3

grid start from 1. Henceforth, I will refer to an image at row i and column j by saying image (i, j).

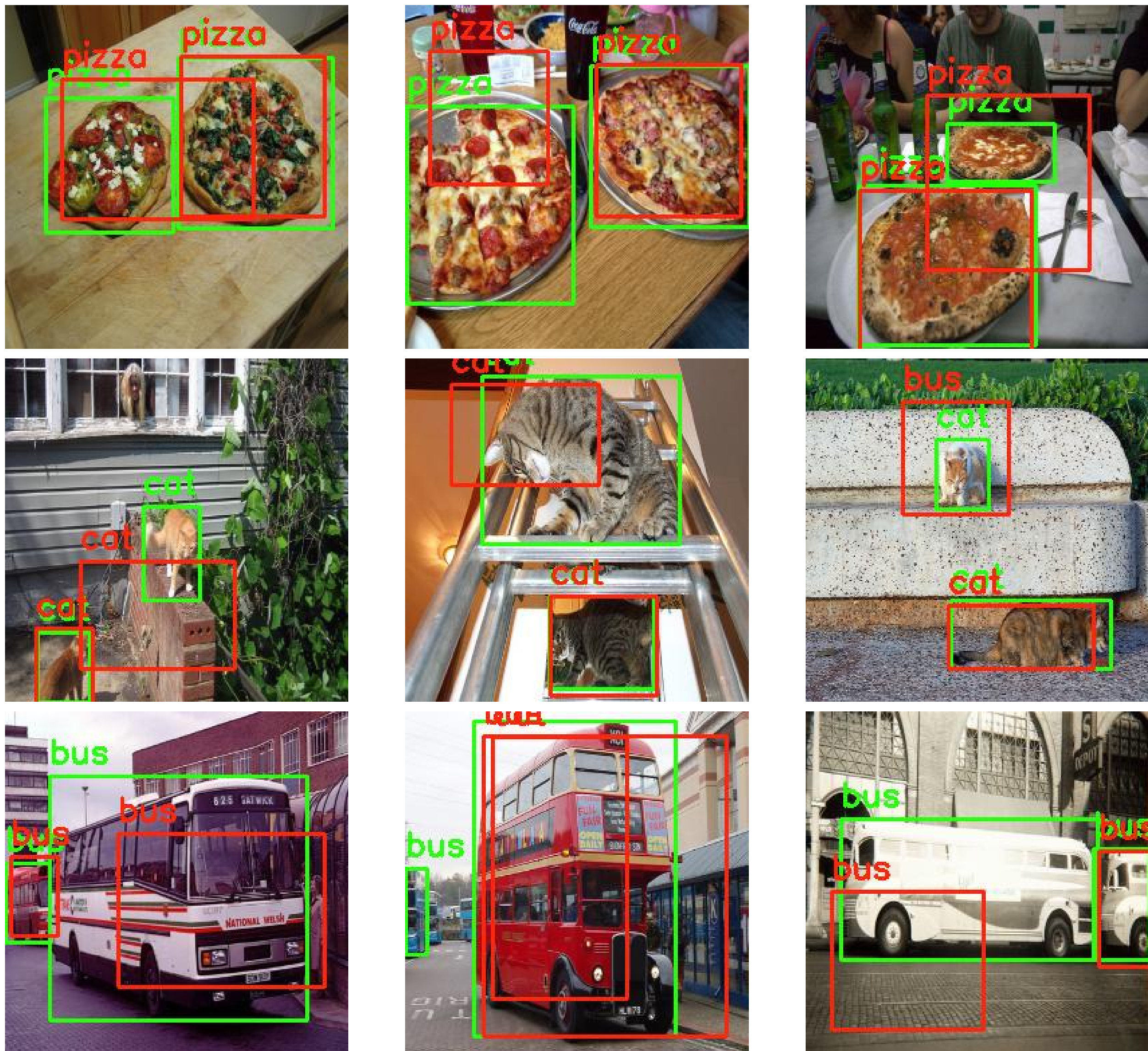


Figure 14: YOLO predictions and ground truths from training dataset

In Figure 14, it seems that model learned the localization and classification of the pizza class better than bus and cat classes. All classifications are correct and bounding boxes are very close to each other. In image $(1, 2)$ and image $(1, 3)$, although predicted bounding boxes for one object may be loose or not overlap very well the ground truth bounding box, they are reasonable and other object was found perfectly. In image $(1, 1)$, although the cell centers of the images are close and anchor boxes are similar, model is able to distinguish the two models belonging to a same class. Also, this shows that the threshold that I set for NMS algorithm is reasonable because if it was too low, then one of the predicted bounding boxes would be removed in image $(1, 1)$. Regarding this, one may ask why there are overlapping bounding boxes in image $(3, 2)$, this is because boxes belong to different classes and as explained previously, NMS algorithm is run on each class separately. Another observation is model learns to detect small and incomplete objects

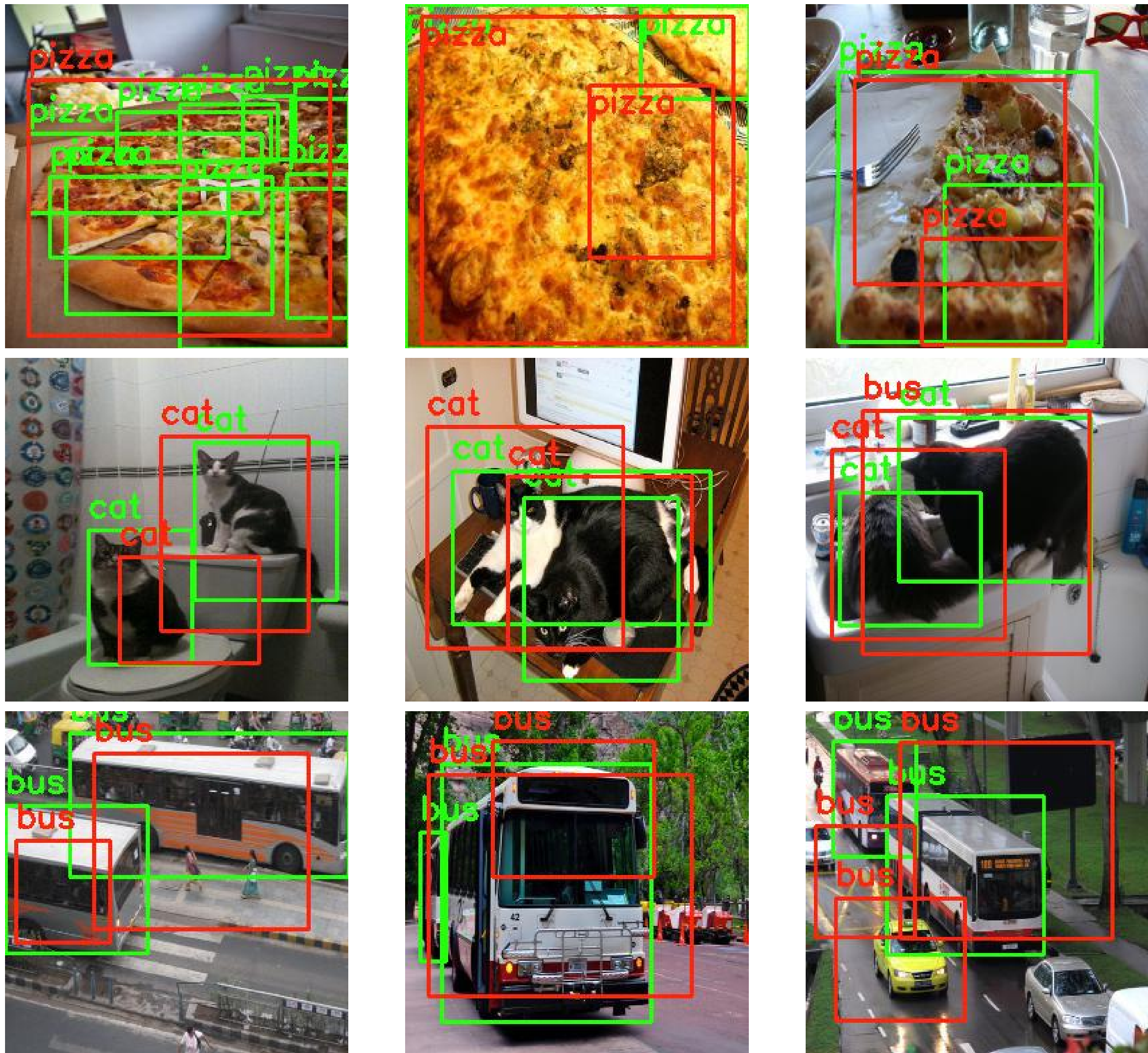


Figure 15: YOLO predictions and ground truths from test dataset

as in images $(3,1)$, $(2,1)$, $(3,3)$, and $(2,2)$. Next I will discuss test data results.

I presented the test (unseen data) results of the model in Figure 15. I tried to put good predictions as well as bad ones. I believe model did well on the following images: $(1,3)$, $(2,1)$, $(2,2)$, $(3,1)$, and $(3,2)$. There is no classification error in these images and localization performances are good considering the fact that model has not seen these images before. In image $(3,2)$, big bounding box covered most of the object very well and there is an extra bounding box, which is also classified as bus. Such incidences are expected because IOU and objectness thresholds are set manually so they are very likely not ideal. And model missed a single very small ground truth, which is also difficult for me to distinguish. In image $(3,3)$, model drawed a bounding box for car, which is not a class in our task, and classified it as a bus. This is not very suprising because car and bus share some features like having tires and being on traffic or road and CNNs are scale invariant, there can be such cases. That problem can be mitigated by either introducing a car class or using extensive

data augmentation techniques as done in [1]. Furthermore, in image $(1, 1)$, there are 8 pizza slices and each of them has bounding boxes. Instead of classifying them separately, my model drawed a large bounding box which covers all pizza objects in it. I think this is acceptable because pizza object definition in the training set does not have clear bounds. For example, in Figure 14, image $(1, 2)$, although the slices of pizza object at the left are distinct, there are no bounding boxes for each of them. Similarly, in test set, image $(1, 3)$, there are two bounding boxes for the whole pizza and just one slice, which implies pizza ground truths are not very clear in the dataset. Thus, we should not expect a good performance for network to make separate bounding boxes of each pizza objects that are close to each other.

4.3 How to improve the performance further?

My main problem was overfitting. As I saved the model whose mean performance on the test set is the best, my final model is not overfitted model but I could not increase the test performance further due to overfitting. I have tried to implement some augmentation techniques and dropout techniques but they did not affect the results significantly. In the original YOLO paper [1], they did extensive augmentations on their data set and the pre-trained their CNN backbone over the entire ImageNet dataset before the training of YOLO. I was going to use the same technique and loaded pre-trained Resnet models but as the datasize is not as large as in the paper, I suspected it could overfit due to large model size and relatively low datasize. I believe, the performance of my model on test samples and training loss plots indicate that my model generalizes the data, makes relatively good and sufficient object detection, and YOLO logic was implemented correctly. My model architecture can be seen on Figure 16.

4.4 How to run the code?

Lastly, I want to explain how to run the code if TA wants to perform data generation, training and inference to see sample predictions. Steps for running the code as follows: **i)** after downloading the COCO dataset from its website, one can run the following command:

```
python dataset.py --coco_dir <path for parent folder of coco>
```

and coco is a folder that contains following folders: test2014, train2014, and annotations2014. They include test, train and annotation files downloaded from COCO webpage. To run the program, same name convention should be adopted. This command will create two dataset folders, which are "train_data" and "test_data". They are pre-processed images from COCO dataset. **ii)** To run the training, following command should be run:

```
python hw6_MehmetBerkSahin.py --coco_dir <path for parent folder of coco>
```

To change the hyperparameters of the network and training, one can check the source code for their names, and make arbitrary changes via command. For example, epochs and `yolo_interval` are 100 and 20 respectively. To change them, one needs to write the following:

```
python hw6_MehmetBerkSahin.py --coco_dir <path for parent folder of coco> --epochs
100 --yolo_interval 30
```

Lastly, **iii)** to make inference and get the sample results, one needs to write the following command:

```
python hw6_MehmetBerkSahin.py --coco_dir <path for parent folder of coco>
--inference True
```

Predictions on test set will be saved to the local disk as "test_pred.jpeg". As I performed the training on NVIDIA A100 GPU, running the program may require a GPU support.

5 Appendix

5.1 Model Summary

```
--------------------------------------------------------------------------
      Layer (type)          Output Shape         Param #     Tr. Param #
==========================================================================
  ReflectionPad2d-1      [1, 3, 262, 262]               0               0
          Conv2d-2       [1, 8, 256, 256]           1,184           1,184
     BatchNorm2d-3       [1, 8, 256, 256]              16              16
            ReLU-4       [1, 8, 256, 256]               0               0
          Conv2d-5     [1, 16, 128, 128]           1,168           1,168
     BatchNorm2d-6     [1, 16, 128, 128]              32              32
            ReLU-7     [1, 16, 128, 128]               0               0
          Conv2d-8       [1, 32, 64, 64]           4,640           4,640
     BatchNorm2d-9       [1, 32, 64, 64]              64              64
           ReLU-10       [1, 32, 64, 64]               0               0
         Conv2d-11       [1, 64, 32, 32]          18,496          18,496
    BatchNorm2d-12       [1, 64, 32, 32]             128             128
           ReLU-13       [1, 64, 32, 32]               0               0
         Conv2d-14      [1, 128, 16, 16]          73,856          73,856
    BatchNorm2d-15      [1, 128, 16, 16]             256             256
           ReLU-16      [1, 128, 16, 16]               0               0
          Block-17      [1, 128, 16, 16]         295,680         295,680
          Block-18      [1, 128, 16, 16]         295,680         295,680
          Block-19      [1, 128, 16, 16]         295,680         295,680
          Block-20      [1, 128, 16, 16]         295,680         295,680
          Block-21      [1, 128, 16, 16]         295,680         295,680
         Linear-22            [1, 11520]     377,498,880     377,498,880
    BatchNorm1d-23            [1, 11520]          23,040          23,040
           ReLU-24            [1, 11520]               0               0
         Linear-25             [1, 5760]      66,360,960      66,360,960
==========================================================================
Total params: 445,461,120
Trainable params: 445,461,120
Non-trainable params: 0
--------------------------------------------------------------------------
```

Figure 16: My YOLO model summary

5.2 Source Code

5.3 dataset.py

```
import torch
from pycocotools.coco import COCO
import pickle
import argparse
import os
import numpy as np
import random
import torch.nn as nn
import cv2
import torchvision.transforms as tvt
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from skimage import io
from skimage.transform import resize
import skimage


class YoloDataset(nn.Module):
    def __init__(self, coco, catIds, data_path, coco_inv_labels=None,
                 yolo_interval=20, img_size=256, max_obj=14, anchor_num=5, transform=True,
                 train=True):
        super(YoloDataset, self).__init__()
        # dataset
        self.coco = coco
        self.catIds = catIds
        self.data_path = data_path
        self.train = train
        self.img_size = img_size
        self.coco_inv_labels = coco_inv_labels
        # pre-processing
        self.transform = tvt.Compose([tvt.ToTensor()]) if (transform != None) else None
        # yolo parameters
        self.yolo_interval = yolo_interval
        self.num_cells_width = self.num_cells_height = img_size // self.yolo_interval
        self.anchor_num = anchor_num
        self.max_obj = max_obj
        # dataset generator
        self.folder_name = "train_data" if train else "test_data"
        self.data = self.data_generator() if not os.path.exists(self.folder_name + ".pkl") \
            else pickle.load(open(self.folder_name + ".pkl", "rb"))
        self.file_list = os.listdir(self.folder_name)

    def __len__(self):
        return len(self.file_list)

    def __getitem__(self, item):
        # get the image
        I = io.imread(os.path.join(self.folder_name, self.file_list[item]))
        image = np.uint8(I)
        if self.transform != None:
            image = self.transform(image)
        # get annotations
        ground_truths = self.data[self.file_list[item]]
        return image, ground_truths

    def yolo_extractor(self, anns, x_scale, y_scale):
        yolo_tensor = np.zeros((self.num_cells_width * self.num_cells_height, self.anchor_num, 8)) # you may need to add 9th element later
        # save cell index, anchor num and label
        cell_anc_cat = np.zeros((self.max_obj, 4))
        cell_anc_cat[:,-2:] = 13 # 13 means there is no object
        for i, ann in enumerate(anns):
            # take the bounding box
            class_idx = self.coco_inv_labels[ann['category_id']]
            [x, y, w, h] = ann['bbox']
            # scale the images due to resizing
            [x, y, w, h] = [x * x_scale, y * y_scale, w * x_scale, h * y_scale]
            # bbox center
            x_center, y_center = y + h/2, x + w/2
            # cell index (i, j)
            row_cell_idx = min(x_center // self.yolo_interval, self.num_cells_height - 1) # ith row
            col_cell_idx = min(y_center // self.yolo_interval, self.num_cells_width - 1) # jth column
            # bounding box scale
            bw = w / self.yolo_interval
            bh = h / self.yolo_interval
            # cell center
            cell_i_center = row_cell_idx * self.yolo_interval + self.yolo_interval / 2
            cell_j_center = col_cell_idx * self.yolo_interval + self.yolo_interval / 2
            # calculate difference between centers
            dx = (x_center - cell_i_center) / self.yolo_interval
            dy = (y_center - cell_j_center) / self.yolo_interval
            # aspect ratio
            AR = h / w
            if AR <= 0.2:           anc_idx = 0
            if 0.2 < AR <= 0.5:     anc_idx = 1
            if 0.5 < AR <= 1.5:     anc_idx = 2
            if 1.5 < AR <= 4.0:     anc_idx = 3
```

```
            if 4.0 < AR:              anc_idx = 4
            yolo_vector = np.array([1, dx, dy, bh, bw, 0, 0, 0])
            yolo_vector[5 + class_idx] = 1
            # save the yolo_vector to yolo_tensor
            yolo_tensor[int(row_cell_idx * self.num_cells_width + col_cell_idx), anc_idx, :] = yolo_vector
            cell_anc_cat[i,:2] = (row_cell_idx, col_cell_idx)
            cell_anc_cat[i, 2] = anc_idx
            cell_anc_cat[i, 3] = class_idx

        return cell_anc_cat, yolo_tensor

    def data_generator(self):
        # keeps the file names as keys and annotations as values
        data = {}

        for cat_id in catIds:
            imgIds = self.coco.getImgIds(catIds=cat_id)
            for img_id in imgIds:
                # get annotations
                annIds = self.coco.getAnnIds(imgIds=img_id, catIds=cat_id,
                                        iscrowd=False,
                                        areaRng=[64*64, float('inf')])
                # load annotations
                anns = self.coco.loadAnns(annIds)
                if len(anns) < 1:
                    continue

                # load the image with resizing
                img = self.coco.loadImgs(img_id)[0]
                I = io.imread(os.path.join(self.data_path, img['file_name']))
                if len(I.shape) == 2:
                    I = skimage.color.gray2rgb(I)
                img_h, img_w = I.shape[0], I.shape[1]
                I = resize(I, (self.img_size, self.img_size), anti_aliasing=True, preserve_range=True)
                image = np.uint8(I)
                # scale annotation bounding boxes
                x_scale, y_scale = self.img_size / img_w, self.img_size / img_h
                # get yolo_tensor and other annotations
                cell_anc_cat, yolo_tensor = self.yolo_extractor(anns, x_scale, y_scale)
                data[img['file_name']] = {"cell_idx" : cell_anc_cat[:,:2],
                                          "anchor_idx" : cell_anc_cat[:,2],
                                          "label" : cell_anc_cat[:,3],
                                          "yolo_tensor" : yolo_tensor}
                # save the image
                if self.train:
                    io.imsave(os.path.join("train_data", img['file_name']), image)
                else:
                    io.imsave(os.path.join("test_data", img['file_name']), image)

        print("data generation finished and dictionary was saved.")
        # save the data as .pkl
        pickle.dump(data, open(self.folder_name + '.pkl', 'wb'))
        return data

    def plot_images(self, data_loader, sample_num=3):
        class_list = ['bus', 'cat', 'pizza']  # class list

        class_counter = {idx: 0 for idx in range(3)}

        # display the predictions in a figure
        fig, axs = plt.subplots(3, sample_num)
        #fig.tight_layout()

        img_counter = 0
        for k, data in enumerate(data_loader):
            if class_counter[0] == sample_num and class_counter[1] == sample_num and class_counter[2] == sample_num:
                break
            imgs, gts = data
            yolo_tensor, anchor_idx = gts['yolo_tensor'].numpy(), gts['anchor_idx'].numpy()
            cell_idx, labels = gts['cell_idx'].numpy(), gts['label'].numpy()

            yolo_tensor = yolo_tensor.reshape(self.num_cells_height, self.num_cells_width, self.anchor_num, 8)

            # prepare image for display
            img = np.uint8(imgs[0].numpy() * 255)
            img = img.transpose((1, 2, 0))
            img = np.ascontiguousarray(img)

            label = int(labels[0, 0].item())
            if class_counter[label] >= sample_num:
                continue
            else:
                class_counter[label] += 1
            class_name = class_list[label]

            obj_idxs = np.where(anchor_idx[0] != 13)[0]
            # iterate through objects
            for obj_idx in range(len(obj_idxs)):
                # ground truths
                label = int(labels[0, obj_idx].item())

                row_cell_idx, col_cell_idx = cell_idx[0, obj_idx]
```

```
                anc_idx = anchor_idx[0, obj_idx]

                [row_cell_idx, col_cell_idx, anc_idx] = list(map(lambda x: int(x.item()), [row_cell_idx, col_cell_idx, anc_idx]))
                # pick the yolo_vector
                yolo_vector = yolo_tensor[row_cell_idx, col_cell_idx, anc_idx]

                # calculate ground truth bbox size
                h = yolo_vector[3].item() * self.yolo_interval
                w = yolo_vector[4].item() * self.yolo_interval
                # calculate cell centers
                cell_i_center = row_cell_idx * self.yolo_interval + self.yolo_interval / 2
                cell_j_center = col_cell_idx * self.yolo_interval + self.yolo_interval / 2
                # calculate the center of gt bbox
                x_center = yolo_vector[1].item() * self.yolo_interval + cell_i_center
                y_center = yolo_vector[2].item() * self.yolo_interval + cell_j_center
                [x1, y1, x2, y2] = [round(y_center - w / 2), round(x_center - h / 2), round(y_center + w / 2),
                                    round(x_center + h / 2)]
                # draw the bounding box
                img = cv2.rectangle(img, (round(x1), round(y1)),
                                    (round(x2), round(y2)),
                                    (36, 256, 12), 2)

                img = cv2.putText(img, class_list[label], (round(x1), round(y1 - 10)), cv2.FONT_HERSHEY_SIMPLEX,
                                  0.8, (36, 256, 12), 3) #0.8

            # plot the image
            row, col = img_counter // sample_num, img_counter % sample_num
            axs[row, col].imshow(img)
            axs[row, col].axis('off')
            axs[row, col].set_title(f"class: {class_name}", size=8)
            img_counter += 1

        if self.train:
            name = "train_samples"
        else:
            name = "test_samples"
        plt.savefig(f"{name}.jpeg")
        print("Predictions are plotted and figure is saved!")

if __name__ == "__main__":
    # important directories

    parser = argparse.ArgumentParser()
    parser.add_argument("--coco_dir", default="/Users/berksahin/Desktop",
                        help="parent directory of coco dataset")

    args = parser.parse_args()

    coco_dir = args.coco_dir
    train_dir = os.path.join(coco_dir, "coco/train2014")
    test_dir = os.path.join(coco_dir, "coco/test2014")
    ann_dir = os.path.join(coco_dir, "coco/annotations2014")

    class_list = ['bus', 'cat', 'pizza']
    train_json = 'instances_train2014.json'
    test_json = 'instances_val2014.json'

    if not os.path.exists("train_data"):
        os.mkdir("train_data")
    if not os.path.exists("test_data"):
        os.mkdir("test_data")

    seed = 16 # 7
    np.random.seed(seed)
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmarks = False

    # train and test COCOs
    coco_train = COCO(os.path.join(os.path.join(coco_dir, ann_dir), train_json))
    coco_test = COCO(os.path.join(os.path.join(coco_dir, ann_dir), test_json))
    # mapping from coco labels to my labels
    coco_inv_labels = {}
    catIds = coco_train.getCatIds(catNms=class_list)
    for idx, catId in enumerate(sorted(catIds)):
        coco_inv_labels[catId] = idx

    pickle.dump(coco_inv_labels, open('inv_map.pkl', 'wb'))
    print("Inverse map saved.")

    train_dataset = YoloDataset(coco=coco_train, catIds=catIds, data_path=os.path.join(coco_dir, train_dir),
                                coco_inv_labels=coco_inv_labels, train=True)
    test_dataset = YoloDataset(coco=coco_test, catIds=catIds, data_path=os.path.join(coco_dir, test_dir),
                               coco_inv_labels=coco_inv_labels, train=False)
    train_loader = DataLoader(train_dataset, batch_size=1, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=1, shuffle=True)
    print(f"Length of the train dataset: {len(train_dataset)}")

    # plot samples from train dataset
```

```
    train_dataset.plot_images(train_loader)
    print("samples from train dataset were saved!")
    # plot samples from test dataset
    test_dataset.plot_images(test_loader)
    print("samples from test dataset were saved!")
```

5.4 model.py

```
import torch.nn as nn
import torch.nn.functional as F
import torch
from torch import optim
import time
import numpy as np
from dataset import YoloDataset
import matplotlib.pyplot as plt
from utils import IoU
import cv2
from torchvision.ops import nms, remove_small_boxes
from torchvision.ops import box_iou
from skimage import io
from pytorch_model_summary import summary


# RESIDUAL BLOCK
class Block(nn.Module):

    def __init__(self, width, learnable_res=False):
        super(Block, self).__init__()
        # 2 convolutional layers with batchnorm
        self.conv = nn.Sequential(nn.Conv2d(width, width, 3, 1, 1),
                                  nn.BatchNorm2d(width),
                                  nn.ReLU(inplace=True),
                                  nn.Conv2d(width, width, 3, 1, 1),
                                  nn.BatchNorm2d(width))
        # for learnable skip-connections/residuals
        self.learnable_res = learnable_res
        if self.learnable_res:
            self.res_conv = nn.Sequential(
                nn.Conv2d(width, width, 3, 1, 1),
                nn.BatchNorm2d(width)
            )

    def forward(self, x):
        out = self.conv(x)  # pass through CNN
        if self.learnable_res:
            out += self.res_conv(x)  # pass through learnable res
        else:
            out += x  # skip-connection
        return F.relu(out)  # ReLU


# ENTIRE NETWORK
class YoloNet(nn.Module):
    def __init__(self, in_channels=3, width=8, n_blocks=5, learnable_res=False,
                 max_col_cell=12, max_row_cell=12, anchor_num=5, yolo_interval=20,
                 threshold=0.2, iou_ths=0.5, lamb_obj=5, lamb_noobj=.5):
        assert (n_blocks >= 0)
        super(YoloNet, self).__init__()
        # hyperparamters
        self.threshold = threshold
        self.lamb_obj = lamb_obj
        self.lamb_noobj = lamb_noobj
        self.iou_ths = iou_ths
        # output size
        self.out_dim = max_col_cell * max_row_cell * anchor_num * 8
        self.max_row_cell = max_row_cell
        self.max_col_cell = max_col_cell
        self.yolo_interval = yolo_interval
        self.anchor_num = anchor_num
        # base model
        model = [nn.ReflectionPad2d(3),
                 nn.Conv2d(in_channels, width, kernel_size=7,
                           padding=0),
                 nn.BatchNorm2d(width),
                 nn.ReLU(True)]

        # downsampling layers
        n_down = 4
        mult = 0
        for k in range(n_down):
            expansion = 2 ** k
            model += [nn.Conv2d(width * expansion, width * expansion * 2,
                                kernel_size=3, stride=2, padding=1),
                      nn.BatchNorm2d(width * expansion * 2),
                      nn.ReLU()] # relu added
            mult = width * expansion * 2
        # add residual blocks
        for i in range(n_blocks):
            model += [Block(mult, learnable_res)]
```

```
        # put the objects in list to nn.Sequential
        self.model = nn.Sequential(*model)
        # classifier head
        self.class_head = nn.Sequential(
            nn.Linear(32768, 11520),
            nn.BatchNorm1d(11520),
            nn.ReLU(True),
            nn.Linear(11520, self.out_dim)
        )

    def forward(self, x):
        out = self.model(x)
        out = torch.flatten(out, 1)
        out = self.class_head(out)
        return out

    def run_train(self, train_loader, test_loader, epochs=50, lr=1e-3, betas=(0.9, 0.99)):
        # load model to device (cpu or gpu)
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self = self.to(device)
        # loss functions for binary class, mse, multi class
        criterion1 = nn.BCELoss()
        criterion2 = nn.MSELoss()
        criterion3 = nn.CrossEntropyLoss()
        # choose optimization method
        optimizer = optim.Adam(self.parameters(), lr, betas)
        start_time = time.perf_counter()
        train_loss_hist = {"bce" : [], "mse" : [], "ce" : []}
        test_loss_hist = {"bce" : [], "mse" : [], "ce" : []}
        min_loss = 100000000000
        print("Training is running...")
        # training loop
        for epoch in range(epochs):
            # losses that will be reported at each epoch
            run_loss_bce = 0.0
            run_loss_ce = 0.0
            run_loss_mse = 0.0
            self.train()
            for data in train_loader:
                # extract data
                imgs, gts = data
                yolo_tensor, label = gts['yolo_tensor'], gts['label']
                # load them to gpu or cpu
                imgs = imgs.to(device)
                yolo_tensor = yolo_tensor.float().to(device)
                # forward pass
                optimizer.zero_grad()
                output = self(imgs)
                output = output.view(yolo_tensor.shape)

                # CALCULATE TOTAL LOSS
                loss = torch.tensor(0.0, requires_grad=True).float().to(device)
                objectness = yolo_tensor[:,:,:,0] # objectness ground truths
                yolo_idx = torch.nonzero(objectness) # consider cell&anch that have objects

                # CALCULATE BCE LOSS
                # calculate binary cross entropy for no object cases
                tmp = torch.nonzero(objectness == 0)
                no_obj_pred = output[tmp[:,0], tmp[:,1], tmp[:,2]]
                no_obj_gt = yolo_tensor[tmp[:,0], tmp[:,1], tmp[:,2]]
                loss_no_obj = criterion1(nn.Sigmoid()(no_obj_pred[:,0]), no_obj_gt[:,0])
                loss += self.lamb_noobj * loss_no_obj

                # calculate binary cross entropy for objects cases
                obj_pred = output[yolo_idx[:,0], yolo_idx[:,1], yolo_idx[:,2]]
                obj_gt = yolo_tensor[yolo_idx[:,0], yolo_idx[:,1], yolo_idx[:,2]]
                loss_obj = criterion1(nn.Sigmoid()(obj_pred[:,0]), obj_gt[:,0])
                loss += loss_obj

                # save the loss for bce
                run_loss_bce += loss_no_obj.item() + loss_obj.item()

                # CALCULATE MSE LOSS
                yolo_tensor = yolo_tensor.view(-1, self.max_row_cell, self.max_col_cell, self.anchor_num, 8)
                output = output.view(yolo_tensor.shape)
                # pull cell numbers of objects
                objects = torch.nonzero(label != 13) # only include real objects
                cell_nos = gts['cell_idx'][objects[:,0], objects[:,1]].type(torch.long)
                anchor_nos = gts['anchor_idx'][objects[:,0], objects[:,1]].type(torch.long)
                #anchor_nos = gts['anchor_idx'][objects[:,0], objects[:,1]]
                output = output[objects[:,0], cell_nos[:,0], cell_nos[:,1], anchor_nos]
                yolo_tensor = yolo_tensor[objects[:,0], cell_nos[:,0], cell_nos[:,1], anchor_nos]
                # calculate mse loss
                mse_loss_obj = criterion2(output[:,1:5], yolo_tensor[:,1:5])
                loss += self.lamb_obj * mse_loss_obj
                run_loss_mse += mse_loss_obj.item()
                # calculate multi class cross entropy loss
                labels = torch.argmax(yolo_tensor[:,5:], dim=1)
                tmp = criterion3(output[:,5:], labels)
                loss += tmp
                run_loss_ce += tmp.item()
                # backward pass
```

```
            loss.backward()
            optimizer.step()

        # calculate mean losses
        run_loss_bce /= len(train_loader)
        run_loss_mse /= len(train_loader)
        run_loss_ce /= len(train_loader)
        total_loss = (run_loss_bce + run_loss_mse + run_loss_ce) / 3 # equal weights
        # save mean losses
        train_loss_hist["bce"].append(run_loss_bce)
        train_loss_hist["mse"].append(run_loss_mse)
        train_loss_hist["ce"].append(run_loss_ce)
        # report the results
        print("*"*15 + "TRAIN" + "*"*15)
        print(f"[EPOCH {epoch+1}/{epochs}] Total Mean Loss: {round(total_loss, 5)}")
        print(f"[EPOCH {epoch+1}/{epochs}] BCE Mean Loss: {round(run_loss_bce, 5)}")
        print(f"[EPOCH {epoch+1}/{epochs}] MSE Loss: {round(run_loss_mse, 5)}")
        print(f"[EPOCH {epoch+1}/{epochs}] CE Mean Loss: {round(run_loss_ce, 5)}")

        # evaluation of the model (to check overfitting)
        run_loss_bce = 0.0
        run_loss_ce = 0.0
        run_loss_mse = 0.0
        self.eval()
        for data in test_loader:
            # extract data
            imgs, gts = data
            yolo_tensor, label = gts['yolo_tensor'], gts['label']
            # load them to gpu or cpu
            imgs = imgs.to(device)
            yolo_tensor = yolo_tensor.float().to(device)
            # forward pass
            output = self(imgs)
            output = output.view(yolo_tensor.shape)
            # total loss
            loss = torch.tensor(0.0, requires_grad=True).float().to(device)
            objectness = yolo_tensor[:,:,:,0] # consider cell&anch that have no objects
            yolo_idx = torch.nonzero(objectness) # consider cell&anch that have objects
            # CALCULATE BCE LOSS
            # calculate binary cross entropy for no object cases
            tmp = torch.nonzero(objectness == 0)
            no_obj_pred = output[tmp[:,0], tmp[:,1], tmp[:,2]]
            no_obj_gt = yolo_tensor[tmp[:,0], tmp[:,1], tmp[:,2]]
            loss_no_obj = criterion1(nn.Sigmoid()(no_obj_pred[:,0]), no_obj_gt[:,0])
            # calculate binary cross entropy for objects cases
            obj_pred = output[yolo_idx[:,0], yolo_idx[:,1], yolo_idx[:,2]]
            obj_gt = yolo_tensor[yolo_idx[:,0], yolo_idx[:,1], yolo_idx[:,2]]
            loss_obj = criterion1(nn.Sigmoid()(obj_pred[:,0]), obj_gt[:,0])
            # save the loss for bce
            run_loss_bce += loss_no_obj.item() + loss_obj.item()
            # CALCULATE MSE LOSS
            yolo_tensor = yolo_tensor.view(-1, self.max_row_cell, self.max_col_cell, self.anchor_num, 8)
            output = output.view(yolo_tensor.shape)
            # pull cell numbers of objects
            objects = torch.nonzero(label != 13) # only include real objects
            cell_nos = gts['cell_idx'][objects[:,0], objects[:,1]].type(torch.long)
            anchor_nos = gts['anchor_idx'][objects[:,0], objects[:,1]].type(torch.long)
            #anchor_nos = gts['anchor_idx'][objects[:,0], objects[:,1]]
            output = output[objects[:,0], cell_nos[:,0], cell_nos[:,1], anchor_nos]
            yolo_tensor = yolo_tensor[objects[:,0], cell_nos[:,0], cell_nos[:,1], anchor_nos]
            # calculate mse loss
            mse_loss_obj = criterion2(output[:,1:5], yolo_tensor[:,1:5])
            run_loss_mse += mse_loss_obj.item()
            # calculate multi class cross entropy loss
            labels = torch.argmax(yolo_tensor[:,5:], dim=1)
            tmp = criterion3(output[:,5:], labels)
            run_loss_ce += tmp.item()

        # calculate mean losses
        run_loss_bce /= len(test_loader)
        run_loss_mse /= len(test_loader)
        run_loss_ce /= len(test_loader)
        total_loss = (run_loss_bce + run_loss_mse + run_loss_ce) / 3 # equal weights
        # save mean losses
        test_loss_hist["bce"].append(run_loss_bce)
        test_loss_hist["mse"].append(run_loss_mse)
        test_loss_hist["ce"].append(run_loss_ce)
        # report the results
        print("*"*15 + "TEST" + "*"*15)
        print(f"[EPOCH {epoch+1}/{epochs}] Total Mean Loss: {round(total_loss, 5)}")
        print(f"[EPOCH {epoch+1}/{epochs}] BCE Mean Loss: {round(run_loss_bce, 5)}")
        print(f"[EPOCH {epoch+1}/{epochs}] MSE Loss: {round(run_loss_mse, 5)}")
        print(f"[EPOCH {epoch+1}/{epochs}] CE Mean Loss: {round(run_loss_ce, 5)}")

        if total_loss < min_loss:
            min_loss = total_loss
            # save the best model
            torch.save(self.state_dict(), "best_model")
            print("Best model has been saved!")

    # save the last model Y
    torch.save(self.state_dict(), "last_model")
```

```
        print("last model has been saved!")


        return {"train": train_loss_hist,
                "test" : test_loss_hist}

    def bbox_to_corners(self, indices, yolo_tensor, device):

        [dx, dy, h, w] = [yolo_tensor[:,i] for i in range(1,5)]

        h *= self.yolo_interval
        w *= self.yolo_interval

        row_cell_idx = indices // self.max_col_cell
        col_cell_idx = indices % self.max_col_cell

        cell_i_center = row_cell_idx * self.yolo_interval + self.yolo_interval/2
        cell_j_center = col_cell_idx * self.yolo_interval + self.yolo_interval/2

        # broadcast
        x_center = cell_i_center + dx * self.yolo_interval
        y_center = cell_j_center + dy * self.yolo_interval

        x1 = (y_center - w/2).unsqueeze(dim=1)
        y1 = (x_center - h/2).unsqueeze(dim=1)
        x2 = (y_center + w/2).unsqueeze(dim=1)
        y2 = (x_center + h/2).unsqueeze(dim=1)

        bbox = torch.cat((yolo_tensor[:,0].unsqueeze(dim=1), x1, y1, x2, y2, yolo_tensor[:,5:]), dim=1).to(device)

        return bbox

    def inference(self, model_path, test_loader, sample_num=2):
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # set device
        class_list = ['bus', 'cat', 'pizza'] # class list
        # load model
        self.load_state_dict(torch.load(model_path))
        self.to(device)
        self.eval()
        # set the figure and axes for display
        fig, axs = plt.subplots(3, sample_num)
        fig.tight_layout()
        counter = 0
        # iterate through data loader, batch size = 1
        for data in test_loader:
            # display only sample_num number of images per
            if counter > 3 * sample_num - 1:
                break
            # extract the annotations and images
            imgs, gts = data
            yolo_tensor, anchor_idx = gts['yolo_tensor'], gts['anchor_idx']
            cell_idx, labels = gts['cell_idx'], gts['label']

            # prepare image for display
            img = np.uint8(imgs[0].numpy() * 255)
            img = img.transpose((1, 2, 0))
            img = np.ascontiguousarray(img)

            # load them to gpu or cpu
            imgs = imgs.to(device)
            yolo_tensor = yolo_tensor.float().to(device)
            output = self(imgs)
            # reshape the prediction and labels to (num. row cell, num. col cell, anchor id, 8)
            output = output.view(yolo_tensor.shape)
            yolo_tensor = yolo_tensor.view(self.max_row_cell, self.max_col_cell, self.anchor_num, 8)
            # forward pass for inference
            yolo_idx = torch.nonzero(anchor_idx[0] != 13)
            if len(yolo_idx) == 1:
                continue
            # iterate through objects to display ground truths
            for obj_idx in range(len(yolo_idx)):
                # extract ground truths for each object
                label = int(labels[0, obj_idx].item())
                row_cell_idx, col_cell_idx = cell_idx[0, obj_idx]
                anch_idx = anchor_idx[0, obj_idx]
                [row_cell_idx, col_cell_idx, anch_idx] = list(map(lambda x: int(x.item()), [row_cell_idx, col_cell_idx, anch_idx]))
                # relevant yolo vector
                yolo_vector = yolo_tensor[row_cell_idx, col_cell_idx, anch_idx]
                # calculate ground truth bbox size
                h = yolo_vector[3].item() * self.yolo_interval
                w = yolo_vector[4].item() * self.yolo_interval
                # calculate cell centers
                cell_i_center = row_cell_idx * self.yolo_interval + self.yolo_interval / 2
                cell_j_center = col_cell_idx * self.yolo_interval + self.yolo_interval / 2
                # calculate the center of gt bbox
                x_center = yolo_vector[1].item() * self.yolo_interval + cell_i_center
                y_center = yolo_vector[2].item() * self.yolo_interval + cell_j_center
                [x1, y1, x2, y2] = [round(y_center-w/2), round(x_center-h/2), round(y_center+w/2), round(x_center+h/2)]
                # draw the bounding box
                img = cv2.rectangle(img, (round(x1), round(y1)),
                                    (round(x2), round(y2)),
                                    (36, 256, 12), 2)
```

```
                # draw its class name
                img = cv2.putText(img, class_list[label], (round(x1), round(y1 - 10)), cv2.FONT_HERSHEY_SIMPLEX,
                                                    0.8, (36, 256, 12), 2)
            # objectness prediction
            output[0,:,:,0] = nn.Sigmoid()(output[0,:,:,0])
            cobj_idx = torch.nonzero(output[0,:,:,0] > self.threshold)
            box_cand = output[0, cobj_idx[:,0], cobj_idx[:,1]]
            # extract corners of the predicted bounding box
            pred_boxes = self.bbox_to_corners(cobj_idx[:,0], box_cand, device)
            bboxes = []
            # predicted categories
            pred_cats = torch.argmax(pred_boxes[:,5:], dim=1)
            # perform non-max supression for each class
            for i in range(3):
                idxs = torch.nonzero(pred_cats == i)
                if idxs.shape[0] != 0:
                    # non-max supression for objectness = 1
                    class_tensors = pred_boxes[idxs[:,0]]
                    indices = nms(boxes=class_tensors[:,1:5], scores=class_tensors[:,0], iou_threshold=self.iou_ths)
                    bboxes.append(class_tensors[indices,:])


            # display predicted bounding boxes
            for obj_idx in range(len(bboxes)):
                yolo_pred = bboxes[obj_idx][0]
                # predicted class
                label = torch.argmax(yolo_pred[5:]).item()
                # bounding box corners
                [x1, y1, x2, y2] = [round(yolo_pred[i].item()) for i in range(1,5)]
                # draw the bounding box
                img = cv2.rectangle(img, (x1, y1),
                                    (x2, y2),
                                    (256, 36, 12), 2)
                # draw predicted class name
                img = cv2.putText(img, class_list[label], (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX,
                                    0.8, (256, 36, 12), 2)

            # plot the image to the corresponding cell
            row, col = counter // sample_num, counter % sample_num
            axs[row, col].imshow(img)
            axs[row, col].axis('off')
            axs[row, col].set_title(f"Prediction {counter+1}", size=12)
            counter += 1
        # save figure
        plt.savefig("test_pred.jpeg")
        print("Predictions are plotted and figure is saved!")

# test code for CNN backbone of YoloNet
if __name__ == "__main__":
    x = torch.rand((8, 3, 256, 256))
    model = YoloNet()
    y = model(x)
    print("Input shape: ", x.shape)
    print("Output shape: ", y.shape)
    # print model summary
    model_sum = summary(model, torch.zeros((1, 3, 256, 256)), show_input=False, show_hierarchical=True)

    file_obj = open("model_sum.txt", "w")
    file_obj.write(model_sum)
    print("Model summary was saved...")

    print("Learnable layers:",  len(list(model.parameters())))
```

5.5 hw6_MehmetBerkSahin.py

```
from dataset import YoloDataset
from model import YoloNet
import argparse
from torch.utils.data import DataLoader
import random
import torch, gc
import numpy as np
import os
from pycocotools.coco import COCO
import pickle

if __name__ == '__main__':

    parser = argparse.ArgumentParser()
    parser.add_argument("--epochs", default=10, type=int, help="number of epochs for training")
    parser.add_argument("--lr", default=1e-3, type=float, help="learning rate")
    parser.add_argument("--yolo_interval", default=20, type=int, help="length of one yolo cell")
    parser.add_argument("--anchors", default=5, type=int, help="number of anchor boxes")
    parser.add_argument("--coco_dir", default="/Users/berksahin/Desktop",
                        help="parent directory of coco dataset")
    parser.add_argument("--batch_size", default=32, type=int, help="size of the minibatch")
    parser.add_argument("--img_size", default=256, type=int, help="size of the images for training")
    parser.add_argument("--threshold", default=0.9, type=float, help="threshold for predicted objectness")
    parser.add_argument("--iou_threshold", default=0.4, type=float, help="iou threshold for nonmax supression")
    parser.add_argument("--lambda1", default=5, type=float, help="lambda for objects (mse)") # lambda_coord
    parser.add_argument("--lambda2", default=.5, type=float, help="lambda for no objects") # lambda_noobj
```

```
    parser.add_argument("--inference", default=False, type=bool, help="open inference mode")
    args = parser.parse_args()
    # take the inputs
    EPOCH = args.epochs
    LR = args.lr
    YOLO_INT = args.yolo_interval
    ANC = args.anchors
    COCO_DIR = args.coco_dir
    BATCH = args.batch_size
    IMG_SIZE = args.img_size
    THRESHOLD = args.threshold
    IOU_THS = args.iou_threshold
    LAMBDA1 = args.lambda1
    LAMBDA2 = args.lambda2
    INFERENCE = args.inference

    # for reproducible results
    seed = 3
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic=True
    torch.backends.cudnn.benchmarks=False

    # data file info
    class_list = ['bus', 'cat', 'pizza']
    train_json = 'instances_train2014.json'
    test_json = "instances_val2014.json"
    # create data directories if doesn't exist
    if not os.path.exists("train_data"):
        os.mkdir("train_data")
    if not os.path.exists("test_data"):
        os.mkdir("test_data")

    # train and test COCOs
    coco_train = COCO(train_json)
    coco_test = COCO(test_json)
    # mapping from coco labels to my labels
    coco_inv_labels = {}
    catIds = coco_train.getCatIds(catNms=class_list)
    for idx, catId in enumerate(sorted(catIds)):
        coco_inv_labels[catId] = idx
    # save inverse maps
    pickle.dump(coco_inv_labels, open('inv_map.pkl', 'wb'))
    print("Inverse map saved!")
    # create custom dataset for training and test/inference
    train_dataset = YoloDataset(coco=coco_train, catIds=catIds, data_path="",
                                coco_inv_labels=coco_inv_labels, train=True)
    print(f"length of the train dataset: {len(train_dataset)}")
    test_dataset = YoloDataset(coco=coco_test, catIds=catIds, data_path="",
                               coco_inv_labels=coco_inv_labels, train=False)
    print(f"length of the test dataset: {len(test_dataset)}")
    # initialize dataloaders
    train_loader = DataLoader(train_dataset, batch_size=BATCH, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=BATCH, shuffle=True)

    # initialize model
    model = YoloNet(threshold=THRESHOLD, iou_ths=IOU_THS, lamb_obj=LAMBDA1, lamb_noobj=LAMBDA2)

    if INFERENCE:
        # do inference and save figures
        test_loader2 = DataLoader(test_dataset, batch_size=1, shuffle=True)
        model.inference("best_model", test_loader2)
    else:
        # start training
        loss = model.run_train(train_loader, test_loader, epochs=EPOCH, lr=LR)
        # save the loss history
        pickle.dump(loss, open("loss_history.pkl", "wb"))
        print("training was completed succesfully and losses were saved!")
```

5.6 utils.py

```
from torchvision.ops import box_iou
import torch
import torch.nn as nn

class IoU(nn.Module):

    def __init__(self, yolo_interval, device, reduction='none', image_size=256):
        super(IoU, self).__init__()
        self.reduction = reduction
        self.yolo_interval = yolo_interval
        self.device = device
        self.image_size = image_size

    def bbox_to_corners(self, cell_nos, dx, dy, h, w):
```

```
        h *= self.yolo_interval
        w *= self.yolo_interval

        row_cell_idx = cell_nos[:,0].unsqueeze(dim=1)
        col_cell_idx = cell_nos[:,1].unsqueeze(dim=1)

        cell_i_center = row_cell_idx * self.yolo_interval + self.yolo_interval/2
        cell_j_center = col_cell_idx * self.yolo_interval + self.yolo_interval/2

        # broadcast
        x_center = cell_i_center.repeat(1, dx.shape[1]).to(self.device) + dx * self.yolo_interval
        y_center = cell_j_center.repeat(1, dy.shape[1]).to(self.device) + dy * self.yolo_interval

        #CHECK THIS TOMORROW
        x1 = (y_center - w/2).unsqueeze(dim=2)
        y1 = (x_center - h/2).unsqueeze(dim=2)
        x2 = (y_center + w/2).unsqueeze(dim=2)
        y2 = (x_center + h/2).unsqueeze(dim=2)

        bbox = torch.cat((x1, y1, x2, y2), dim=2).to(self.device)
        return bbox

    def forward(self, cell_nos, output, target):

        [dx, dy, h, w] = [output[:,:,i] for i in range(1,5)]
        bbox_pred = self.bbox_to_corners(cell_nos, dx, dy, h, w)
        [dx, dy, h, w] = [target[:,i].unsqueeze(dim=1) for i in range(1,5)]
        bbox_gt = self.bbox_to_corners(cell_nos, dx, dy, h, w).squeeze(dim=1)
        # calculate iou score
        results = torch.zeros(bbox_pred.shape[:-1], device=self.device, requires_grad=True)
        idx = torch.arange(0, results.shape[0])
        for i in range(results.shape[1]):
            tmp = box_iou(bbox_pred[:,i,:], bbox_gt)
            results[:,i] = tmp[idx, idx]
        return results
```

References

[1] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.