# ECE 60146 Deep Learning Homework 5

Mehmet Berk Sahin, sahinm@purdue.edu, 34740048

March 7, 2023

## 1 Introduction

The main goal of this homework is to create a **pizza detector** by designing a convolutional neural network (CNN). It's an object detector that finds "bus", "cat" or "pizza" objects in an image and draws their bounding boxes. Specifications of how the dataset was constructed will be explained later. To achieve that task, I considered the following criteria: **i)** to mitigate the vanishing gradient problem that we encountered in the previous homework, we are asked to design our Skip-Connection block for extracting convolutional features of an input image. **ii)** Additional fully connected layers were implemented as a regressor and classifier, which correspond to the bounding box predictor and class predictor respectively. **iii)** To train the regressor part of the network, we are asked to implement Complete Intersection over Union (Complete IoU) Loss. And the final performance is reported as IoU. **iv)** I implemented the logic for training and evaluation of the model.

The structure of this report is as follows. In Section 2, I explained how I used the COCO API and constructed my own dataset for this homework. In Section 3, I explained the implementation of the CNN, training, evaluation, and other important submodules that I created to answer the questions of the homework. In Section 4, I discussed the results and in Section 5, I mentioned the takeaways of this homework. I didn't go through the implementation of every function in detail as some of them were very similar to the functions in past homeworks.

Before explaining the dataset, I want to mention the structure of the modules and how to run the program's main code that answers the questions in the homework. My implementation consists of the following modules: `Dataset.py`, `utils.py`, `train.py`, `network.py`, and `main.py`. Although I will explain them in detail in the next sections, let me explain them briefly. `Dataset.py` includes my custom dataset class with additional functions relevant to data such as data generator and display images. `train.py` consists of a function dedicated to the main training loop of CNN. `network.py` includes the CNN implementations such as Skip-Connection block and the whole network `HW5Net`. `utils.py` includes utility functions like inference, display and update confusion matrix, custom loss definitions, and display the predicted bounding box of the network.

## 2 Dataset Generation

Before embarking on the network and training design, it is crucial to implement a well-designed dataset. Although COCO provides a good API for its users, it is not enough for our specific task and one needs to define a custom dataset for the purpose of this homework. In this section, I explained how I generated my custom dataset to train my CNN for object detection. Similar to

HW4, I downloaded **2014 Train images**, **2014 Val images**, and **2014 Train/Val annotations**. In the generation of my custom train and test/val dataset, I filtered the images such that any image in the dataset meets the following criteria:

- It includes at least one of the following three classes: bus, cat, or pizza.
- It only contains one *dominant* object, which is an object with the largest area exceeding $200 \times 200$ pixels from any aforementioned classes. So, images that have more than two or no dominant objects were not included in the dataset.
- It was resized to $256 \times 256$ with its bounding box.
- Training set was constructed from **2014 Train images** and testing set was constructed from **2014 Val images**.

Considering these factors, I end up with 3789 training images and 1972 testing images, which are roughly 4k training and 2k testing images as indicated in the assignment.

### 2.1 `MyCOCODataset`

I implemented my custom dataset as a `MyCOCODataset` class. Its implementation is given in

```
class MyCOCODataset(nn.Module):
    def __init__(self, data, label_map, train=True):
        super(MyCOCODataset, self).__init__()
        self.files = list(data.keys())
        self.data = data
        self.label_map = label_map

        if train:
            self.path = 'train_data'
        else:
            self.path = 'test_data'

        self.transforms = tvt.Compose([tvt.ToTensor()
                                       #tvt.RandomCrop(224,224)])
                                       ])
    def __len__(self):
        return len(self.files)

    def __getitem__(self, item):
        img_file = self.files[item]
        # get the image with transformations
        img = io.imread(os.path.join(self.path, img_file))
        #img = Image.open(os.path.join(self.path, img_file))
        img = self.transforms(img)
        # get the label
        label = self.label_map[self.data[img_file]['category_id']]
        # get the bounding box
        [x1, y1, w, h] = self.data[img_file]['bbox']
        x2, y2 = x1+w, y1+h
        return {"image": img,
                "label": torch.tensor(label),
                "bbox": torch.tensor([x1/255, y1/255, x2/255, y2/255])}
```

Figure 1: `MyCOCODataset` implementation

It takes three arguments: data, label_map, and train. data is a dictionary whose keys are the name of the image files from the COCO dataset and whose values are dictionaries that consist of bounding box annotations and class labels. I constructed that data structure in `data_generator` function. I am not gonna explain this function because its very similar version was implemented in HW4 and the only difference is images were resized to $256 \times 256$ together with their bounding boxes. Additionally, in Figure 1, when images were pulled, they are normalized to be used in the training. This is important because otherwise, the error of bounding boxes will dominate the training due to very large gradients. Moreover, the label_map argument maps the category ids of

the COCO dataset to the class indices of this homework, which are 0, 1, and 2 for bus, cat, and pizza respectively. Lastly, "__get__()" function returns a dictionary that consists of an image with size $256 \times 256$, its label, and the upper left and lower right corner of its bounding box. In `Dataset.py` module, I also implemented `plot_images()` function that displays sample images from each class. I am not gonna explain it here because it is the same as in HW4 except that I incorporated drawing rectangles and class names of the objects, whose implementation was given in HW5, from `cv2` package.

### 2.2 Sample Images

Sample images that are displayed by `plot_images()` function are given in Figure 2. All of the bounding boxes are correct and each image includes only one dominant object. However, there is


Figure 2: Sample images from each class

one interesting observation that I did. In the fifth pizza image, the bounding box wraps only one slice of pizza, which does not seem like a dominant object because the other part of the pizza is larger than the highlighted object. I thought this may be a bug in my implementation but it is not. I realized that there are 6 bounding boxes for 6 different pizza objects. And only the one that is covered by a bounding box exceeds the $200 \times 200$ limit. So, it obeys the criteria of the dataset, though it affects the training negatively. Because there is also semantic information for pizza on the other side of the image but is not annotated. Therefore, the algorithm will be penalized for putting a bounding box to these parts with high loss. However, its classification performance will get affected less because it looks at the whole image, and the semantic information in the entire

image is preserved. It is still affected because there is part of the network which is shared by both the classifier and bounding box regressor, which will be mentioned in Section 3.

# 3 Implementation

In this section, I will explain the implementation of the main training loop, evaluation, network architectures, and custom loss functions. I want to start by defining the new evaluation metrics that we introduced in this homework and giving their custom class definitions.

## 3.1 Intersection Over Union (IoU)

Although one can calculate the mean squared error between the upper left and lower right coordinates of a bounding box in the training, this is not the best idea as we discussed in class. There can be very bad results when MSE loss looks reasonable. So, it is wiser to use intersection over union metric. Let $B$ and $B^{gt}$ be the predicted and ground truth bounding boxes, then it is defined as

$$IoU = \frac{|B \cap B^{gt}|}{|B \cup B^{gt}|} \tag{1}$$

Note that $|.|$ represents the cardinality of the sets. It is simply the ratio of the overlapping area between two bounding boxes and their total area. When the bounding boxes overlap each other, it is 1. If they are completely separated it is 0. Since there is no built-in function for that to be used in the backward pass in PyTorch, I implemented the `mIoU` custom class. As it inherits the properties of `nn.Module`, it can be used with `backward()` function. I only implemented the

```
class mIoU(nn.Module):
    def __init__(self):
        super(mIoU, self).__init__()

    def forward(self, output, target):
        # for single sample (inference)
        if output.shape[0] == 1:
            return box_iou(output, target.unsqueeze(0))
        # for batches (training)
        else:
            mean_loss = 0
            batch_size = output.shape[0]
            # calculate and return the mean loss (1 - mIoU)
            for i in range(batch_size):
                mean_loss += (1 - box_iou(output[i].unsqueeze(0), target[i].unsqueeze(0)).item()) / batch_size
            return mean_loss
```

Figure 3: mIoU implementation

`forward()` function to be used in both training and inference. If there are more than one sample (batch of images), then each sample is passed to `box_iou` function, which is a built-in function of `torchvision.ops`. Then, IoU loss, which is $1 - IoU$, is calculated for each sample, and their averages are taken and returned. For a single sample case, it only returns the IoU value as it is used in the inference. Although IoU loss seems a reasonable metric, it has some flaws. For example, when two bounding boxes are not intersecting each other it remains 1 no matter what the relative positions of the bounding boxes are. So, gradient descent updates will not help to get the boxes close to each other. To fix that, a more advanced version of IoU was developed and it is called Complete IoU.

## 3.2 Complete IoU

In addition to the intersection over union ratio, this loss function considers the convex hull between two bounding boxes to alleviate the aforementioned problem. Furthermore, to make the convergence

much faster, the ratio of the distance between the center of the boxes and the diagonal of the convex hull and the aspect ratio of the predicted bounding box vis-a-vis the ground truth bounding box were considered. It is defined as follows:

$$CIoULoss = 1 - IoU + \frac{|C - B \cup B^{gt}|}{|C|} + \frac{d^2}{c^2} + \alpha * v \tag{2}$$

where $d$ and $c$ are the distance between the centers of boxes and between the length of the diagonal of the convex hull respectively. And $\alpha$ is the positive trade-off parameter, $v$ is the aspect ratio of bounding boxes and is given as:

$$v = \frac{4}{\pi^2}\left(arctan\frac{w^{gt}}{h^{gt}} - arctan\frac{w}{h}\right)^2 \tag{3}$$

where $w$ and $h$ represent the width and height of the predicted bounding box, and $w^{gt}$ and $h^{gt}$ represent the width and height of the ground truth bounding box. To implement this loss function, I used `complete_box_iou_loss` from `torchvision.ops`. However, as it was not suitable for backward pass, I designed my custom `CompleteIoULoss` class given in Figure 20b. It basically wraps the

```python
class CompleteIOULoss(nn.Module):

    def __init__(self, reduction='none'):
        super(CompleteIOULoss, self).__init__()
        self.reduction = reduction

    def forward(self, output, target):
        loss = complete_box_iou_loss(output, target, self.reduction)
        return loss
```

Figure 4: `CompleteIoULoss implementation`

`complete_box_iou_loss` function with properties of `nn.Module`. I called that function inside the `forward()` function and returned the loss. Other than these two loss functions, cross-entropy, and MSE losses were used in the training and evaluation but since they are not new concepts, I do not discuss them here. They are implemented by calling `nn.CrossEntropyLoss` and `nn.MSELoss` functions. Next, I will explain the architecture of my CNN and its implementation.

### 3.3 Network Architecuture

I have implemented the network architecture classes in my `network.py` module. On my first attempt to do this homework, I implemented ResNet52 from its original paper that is shared in the assignment. I observed that since the regressor part is not as dense as the classifier part, although the test classification accuracy gets above 90%, the performance of bounding boxes was very bad. Since we have 4k data for training, I didn't want to increase the power of the regressor. Instead, I make the classifier more shallow and see whether that network is capable of learning the data. In building that network, I utilized the template provided in the homework and used the Resnet34 blocks in [1]. It is demonstration is given in Figure 6. I implemented that block as `Block` class. The entire CNN architecture with fully connected layers can be seen on the next page.

data.txt

```
----------------------------------------------------------------------------
        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
            Conv2d-7       [1, 32, 64, 64]           4,640           4,640
       BatchNorm2d-8       [1, 32, 64, 64]              64              64
            Conv2d-9       [1, 64, 32, 32]          18,496          18,496
      BatchNorm2d-10       [1, 64, 32, 32]             128             128
           Conv2d-11      [1, 128, 16, 16]          73,856          73,856
      BatchNorm2d-12      [1, 128, 16, 16]             256             256
             Block-13      [1, 128, 16, 16]         295,680         295,680
             Block-14      [1, 128, 16, 16]         295,680         295,680
             Block-15      [1, 128, 16, 16]         295,680         295,680
             Block-16      [1, 128, 16, 16]         295,680         295,680
             Block-17      [1, 128, 16, 16]         295,680         295,680
           Linear-18             [1, 1024]      33,555,456      33,555,456
      BatchNorm1d-19             [1, 1024]           2,048           2,048
             ReLU-20             [1, 1024]               0               0
           Linear-21               [1, 64]          65,600          65,600
      BatchNorm1d-22               [1, 64]             128             128
             ReLU-23               [1, 64]               0               0
           Linear-24                [1, 3]             195             195
           Linear-25             [1, 5196]     170,267,724     170,267,724
      BatchNorm1d-26             [1, 5196]          10,392          10,392
             ReLU-27             [1, 5196]               0               0
           Linear-28             [1, 1024]       5,321,728       5,321,728
      BatchNorm1d-29             [1, 1024]           2,048           2,048
             ReLU-30             [1, 1024]               0               0
           Linear-31              [1, 256]         262,400         262,400
      BatchNorm1d-32              [1, 256]             512             512
             ReLU-33              [1, 256]               0               0
           Linear-34               [1, 64]          16,448          16,448
      BatchNorm1d-35               [1, 64]             128             128
             ReLU-36               [1, 64]               0               0
           Linear-37                [1, 4]             260             260
============================================================================
Total params: 211,083,307
Trainable params: 211,083,307
Non-trainable params: 0
----------------------------------------------------------------------------
```

As can be seen above, it is a dense network but it is shallow compared to the Resnet52 model. Moreover, it has 88 learnable layers, which is higher than what is asked for in homework. Since this is not a hierarchical summary, the heads of the architecture are not distinctive. But, if you look at the `Linear-24` and `Linear-37`, you will see the length of their output are 3 and 4 respectively. The former represents the number of classes and the latter is for the upper left and lower right coordinates of the predicted bounding box. The hierarchical summary of the network is given in Figure 5. The components of the architecture are more apparent in that figure. There are

```
========== Hierarchical Summary ==========

HW5Net(
 (model): Sequential(
  (0): ReflectionPad2d((3, 3, 3, 3)), 0 params
  (1): Conv2d(3, 8, kernel_size=(7, 7), stride=(1, 1)), 1,184 params
  (2): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 16 params
  (3): ReLU(inplace=True), 0 params
  (4): Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)), 1,168 params
  (5): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 32 params
  (6): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)), 4,640 params
  (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 64 params
  (8): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)), 18,496 params
  (9): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 128 params
  (10): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)), 73,856 params
  (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 256 params
  (12): Block(
   (conv): Sequential(
    (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)), 147,584 params
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 256 params
    (2): ReLU(inplace=True), 0 params
    (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)), 147,584 params
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 256 params
   ), 295,680 params
  ), 295,680 params
  (13): Block(
   (conv): Sequential(
    (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)), 147,584 params
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 256 params
    (2): ReLU(inplace=True), 0 params
    (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)), 147,584 params
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 256 params
   ), 295,680 params
  ), 295,680 params
  (14): Block(
   (conv): Sequential(
    (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)), 147,584 params
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 256 params
    (2): ReLU(inplace=True), 0 params
    (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)), 147,584 params
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 256 params
   ), 295,680 params
  ), 295,680 params
  (15): Block(
   (conv): Sequential(
    (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)), 147,584 params
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 256 params
    (2): ReLU(inplace=True), 0 params
    (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)), 147,584 params
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 256 params
   ), 295,680 params
  ), 295,680 params
  (16): Block(
   (conv): Sequential(
    (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)), 147,584 params
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 256 params
    (2): ReLU(inplace=True), 0 params
    (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)), 147,584 params
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 256 params
   ), 295,680 params
  ), 295,680 params
 ), 1,578,240 params
 (class_head): Sequential(
  (0): Linear(in_features=32768, out_features=1024, bias=True), 33,555,456 params
  (1): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 2,048 params
  (2): ReLU(inplace=True), 0 params
  (3): Linear(in_features=1024, out_features=64, bias=True), 65,600 params
  (4): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 128 params
  (5): ReLU(inplace=True), 0 params
  (6): Linear(in_features=64, out_features=3, bias=True), 195 params
 ), 33,623,427 params
 (bbox_head): Sequential(
  (0): Linear(in_features=32768, out_features=5196, bias=True), 170,267,724 params
  (1): BatchNorm1d(5196, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 10,392 params
  (2): ReLU(inplace=True), 0 params
  (3): Linear(in_features=5196, out_features=1024, bias=True), 5,321,728 params
  (4): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 2,048 params
  (5): ReLU(inplace=True), 0 params
  (6): Linear(in_features=1024, out_features=256, bias=True), 262,400 params
  (7): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 512 params
  (8): ReLU(inplace=True), 0 params
  (9): Linear(in_features=256, out_features=64, bias=True), 16,448 params
  (10): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True), 128 params
  (11): ReLU(inplace=True), 0 params
  (12): Linear(in_features=64, out_features=4, bias=True), 260 params
 ), 175,881,640 params
), 211,083,307 params

==========
```

Figure 5: Hierarchical summary of the network

two heads: "bbox_head" and "class_head". They correspond to the bounding box predictor and classifier heads, and they are separate from each other. The rest of the network is shared by these two heads. So, doing backpropagation for the bounding box predictor affects the performance of the classifier. As seen in the hierarchical summary, the name of the network is `HW5Net`. It consists

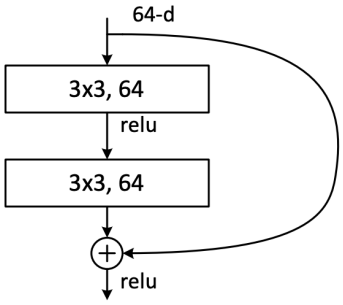



Figure 6: Custom residual block (Block)

of 5 `Block`, which can be seen in Figure 6. Next, I will explain the implementation of residual blocks and the whole network.

### 3.3.1 `Block()`

This class is an implementation of Figure 6. As most of the custom networks in PyTorch, it inherits `nn.Module`. Its implementation is given in Figure 7. Two CNN layers are defined inside `nn.Sequential` object and I put 2D batch normalization between them to enhance the performance. Since it is easy to take the derivate, I used ReLU activation function. Moreover, I added

```
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
```

Figure 7: Block (residual block) implementation

`learnable_res` option so that the network can be tested with residual connections with a convolutional layer rather than a simple summation. If that option is turned off, then input of the block and its output are summed and passed through a ReLU as in Figure 6. Incorporating residual blocks and doing batch normalization mitigate the vanishing gradients problem. The reasons will be discussed in Section 4. Next, I will explain the `HW5Net()` network.

### 3.3.2 `HW5Net()`

I will divide the code into subparts and explain them separately. The first part is given in Figure 8. This part is shared by both regressor and classifier heads. As suggested in the homework template,

```
class HW5Net(nn.Module):

    def __init__(self, in_channels=3, width=8, n_blocks=5, learnable_res=False):
        assert (n_blocks >= 0)
        super(HW5Net, self).__init__()
        # 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),
                      ]
            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)
```

Figure 8: HW5Net().__init__() part 1

I used `nn.ReflectionPad2d` because it does padding with the reflection of boundaries. Thus, the convolution operation will operate on the entire image, which may increase the performance. Then, I used 2D convolution with kernel size $7 \times 7$ to capture the large contextual information in the image. Although this operation reduces the image size, it is not sufficient for fully connected layers and it may result in a very large number of learnable parameters. So, I added 4 downsampling layers, which are convolutional layers and decrease the size of the image. Lastly, to create new abstractions and learn the semantic information in an image, I added 5 residual Blocks. Contrary to HW4, the vanishing gradients problem is prevented thanks to the residual connections. In the last line, all layers including `Block`, `Conv2d`, `BatchNorm2d`, `ReLU`, and `ReflectionPad2d` are put into `nn.Sequential` object, which allows forward propagation to be performed at once.

In the second part, Figure 9, I implemented the regressor and classifier heads. During the

```
        self.class_head = nn.Sequential(
            nn.Linear(32768, 1024),
            nn.BatchNorm1d(1024),
            nn.ReLU(True),
            nn.Linear(1024, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(True),
            nn.Linear(64, 3),
        )
        # bounding box detector head
        self.bbox_head = nn.Sequential(
            nn.Linear(32768, 5196),
            nn.BatchNorm1d(5196),
            nn.ReLU(True),
            nn.Linear(5196, 1024),
            nn.BatchNorm1d(1024),
            nn.ReLU(True),
            nn.Linear(1024, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(True),
            nn.Linear(256, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(True),
            nn.Linear(64, 4))
```

Figure 9: HW5Net().__init__() part 2

experiments, I observed that is harder to get better accuracy for CIoU than classification accuracy.

```python
def forward(self, x):
    out = self.model(x)
    out = torch.flatten(out, 1)
    cat = self.class_head(out)
    bbox = self.bbox_head(out)
    return cat, bbox
```

Figure 10: `HW5Net().forward()` implementation

So, I used a higher number of neurons for the head of the regressor than the classifier. You may notice that the decrease in the number of neurons in the classifier is much faster than the regressor. The latter decreases gradually. Lastly, the implementation of the forward pass is very simple thanks to the utilization of `nn.Sequential` class. It is given in Figure 10. I passed the input through the shared model. Then, it is flattened and put in a suitable format for fully connected head layers. Then, the output of the shared model was passed through the classifier and regressor head and they are returned to be backpropagated later in the training loop. Explaining my custom network, now I can explain the main training loop.

### 3.4 Training Loop

`train()` function was implemented in `train.py` module. As most parts of the training loop are the same as the previous homeworks, I will explain the key points for this homework and you can check the source code for more detail. I believe it is self-explanatory with its comments. First, I

```python
def train(net, num_epochs, batch_size, cIoU=True, model_name='best_model'):

    model = net
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    print(f"training is runing on {device}")
    # load train, test and inv. map dictionaries
    with open('train_data.pkl', 'rb') as f:
        train_dict = pickle.load(f)
    with open('test_data.pkl', 'rb') as f:
        test_dict = pickle.load(f)
    with open('inv_map.pkl', 'rb') as f:
        inv_map = pickle.load(f)
    print("train, test, and inv. map are loaded.")

    # train and test dataset
    train_data = MyCOCODataset(train_dict, inv_map, train=True)
    test_data = MyCOCODataset(test_dict, inv_map, train=False)
    # train and test dataloaders
    train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=True)
    train_size = len(train_loader)
    # train parameters
    criterion1 = nn.CrossEntropyLoss()
    if cIoU:
        criterion2 = CompleteIOULoss('mean')
        loss_name = "CIoU"
    else:
        criterion2 = nn.MSELoss()
        loss_name = "MSE"
```

Figure 11: `train()` implementation part 1

loaded the train data, test data, and label mappings from the COCO dataset to my labels. Then, I initialized the dataloader objects. Note that I only used the test dataset to see the performance of the model at any iteration on the test set. I did not use any sample from the test set for training. After initializing the model, I set the first loss criterion as cross-entropy to be used to train the classifier. And I set the second criterion as either CIoU loss or MSE loss, which is determined by the cIoU argument of the `train()` function. Different from the previous homeworks, as we have two different loss criteria, we should either sum the losses and do one backward pass or do two backward passes. I did the latter as given in Figure 12. Note that I wrote `retain_graph=True`

```python
# calculate loss
cross_loss = criterion1(pred_cat, label)
cross_loss.backward(retain_graph=True)
reg_loss = criterion2(pred_bbox, bbox)
reg_loss.backward()
optimizer.step()
```

Figure 12: `train()` implementation part 2

because if `retain_graph` is left False by default, then part of the graph after the first backward pass will be freed to save memory. Setting it True prevents that. Other parts of the training part are not much different than other homeworks. Since I wanted to see the test performance of the network during training, I added `evaluation()` function under the training loop. That part is given in Figure 13. In the first line, I evaluated the performance of the model on the test

```python
# TEST PART
_, results = evaluation(model, test_loader, criterion1, criterion2, device)
# get the results
run_cross_loss = results["cross-entropy"]
run_reg_loss = results["reg_loss"]
run_iou_loss = results["miou_loss"]
run_acc = results["accuracy"]
# report results
print("*"*12, "TEST", "*"*12)
print(f"[epoch {epoch}/{num_epochs}] test cross entropy loss: {round(run_cross_loss, 4)}")
print(f"[epoch {epoch}/{num_epochs}] test {loss_name}: {round(run_reg_loss, 4)}")
print(f"[epoch {epoch}/{num_epochs}] test mIoU loss: {round(run_iou_loss, 4)}")
print(f"[epoch {epoch}/{num_epochs}] test classification accuracy {round(run_acc*100, 2)}")
print("*"*30)
# save the losses
test_history["cross_loss"].append(run_cross_loss)
test_history["bbox_loss"].append(run_reg_loss)
test_history["acc"].append(run_acc)

model_score = round(run_acc, 3) + (1 - run_reg_loss)
if model_score > max_model_score:
    torch.save(model.state_dict(), model_name)
    print("best model is saved...")
    max_model_score = model_score
```

Figure 13: `train()` implementation part 3

set and get the results. Then, I extracted the losses, reported them on the output terminal, and saved them to the local disk. Lastly, if the summation of the regression and classification losses is larger than the maximum summation from previous iterates, I saved the model to a local disk as the best model. This model was then used to display the sample predicted bounding boxes and classifications. During the experiments, regression loss does not go above 1 so that kind of summation is reasonable for giving approximately equal importance to both metrics. The best way of doing that would be normalizing the losses but as there is no maximum value defined for MSE, normalization does not make sense. However, for the sake of this homework, my implementation is sufficient to save the best model during the training. `train()` function returns the last iterate model, train history, and test history.

### 3.5 Evaluation

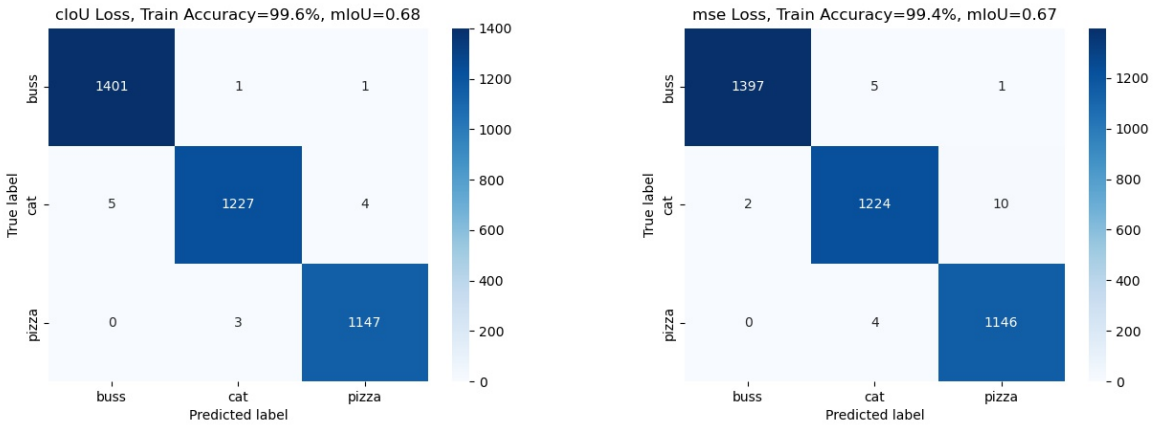
Evaluation of the model on the test set is implemented in `evaluation()`, which is in `utils.py` module. The code is almost the same as the training loop except that the model is in the evaluation mode and it includes a confusion matrix. Thus, the confusion matrix is calculated over the entire test set given the current model whenever this function is called. Since these are the concepts that are covered in the previous homework and since their implementations are the same as in the previous homework, I am not going to explain them again here.

# 4 Results & Discussion

In this section, I present the hyperparameters of my network, the classification, and the mean IoU results for train and test datasets, and discuss them. I chose the Adam optimizer to minimize the loss function with $\beta_1 = 0.9$ and $\beta_2 = 0.99$. To train the classifier, I used cross-entropy loss for multi-class. I conducted two experiments by training the regressor with CIoU and MSE losses. Learning rate, batch size, and the number of epochs were chosen to be $10^{-3}$, 16, and 30 respectively. To obtain the best model and prevent overfitting, I saved the best model with respect to classification accuracy and CIoU losses. In reporting my final model and its results, I did not use learnable residuals because they gave a worse performance than simple residuals with no parameters. I will present and discuss the train and test results by providing their confusion matrices and sample predictions. I run my experiments on NVIDIA A100 GPU.

## 4.1 Train Results

In this section, I discuss the performance of the model over train data. Although it does not tell us much about the generalization performance of the model. It tells us about the capacity of the model. If the model's capacity is sufficient to learn the train data, then eventually, loss functions should approach 0. Confusion matrices over train data for two HW5Net models trained with CIoU and MSE losses can be seen in Figure 14. Henceforth, I will call the model trained with CIoU loss **CIoU model** and the model trained with MSE loss **MSE model**. As can be seen




(a) Confusion matrix of HW5Net with CIoU loss over train data

(b) Confusion matrix of HW5Net with MSE loss over train data

Figure 14: Confusion matrices for CIoU and MSE losses over train data

from the confusion matrices, CIoU model and MSE model achieve 99.6% and 99.4% classification accuracies respectively. This is an indication that both models are capable of learning the data for classification. For the regressor, the ideal case would be $IoU = 1$. When I run the training for 60-70 epochs, IoU approached 0.9 but as the validation performance for IoU does not change, it is sufficient to run training for 30 epochs. Hence, it is clear that the model does not underfit the data and is capable of generalizing it. How well this generalization depends on the model's performance on the test data, which will be elaborate further in the next section. From Figure 14, we observe that models are capable of learning every class because the number of misclassifications is roughly the same. Sample predictions of the CIoU model on the train data can be seen in Figure 15. Red boxes denote predictions of the model and green boxes denote the ground truths. This notation

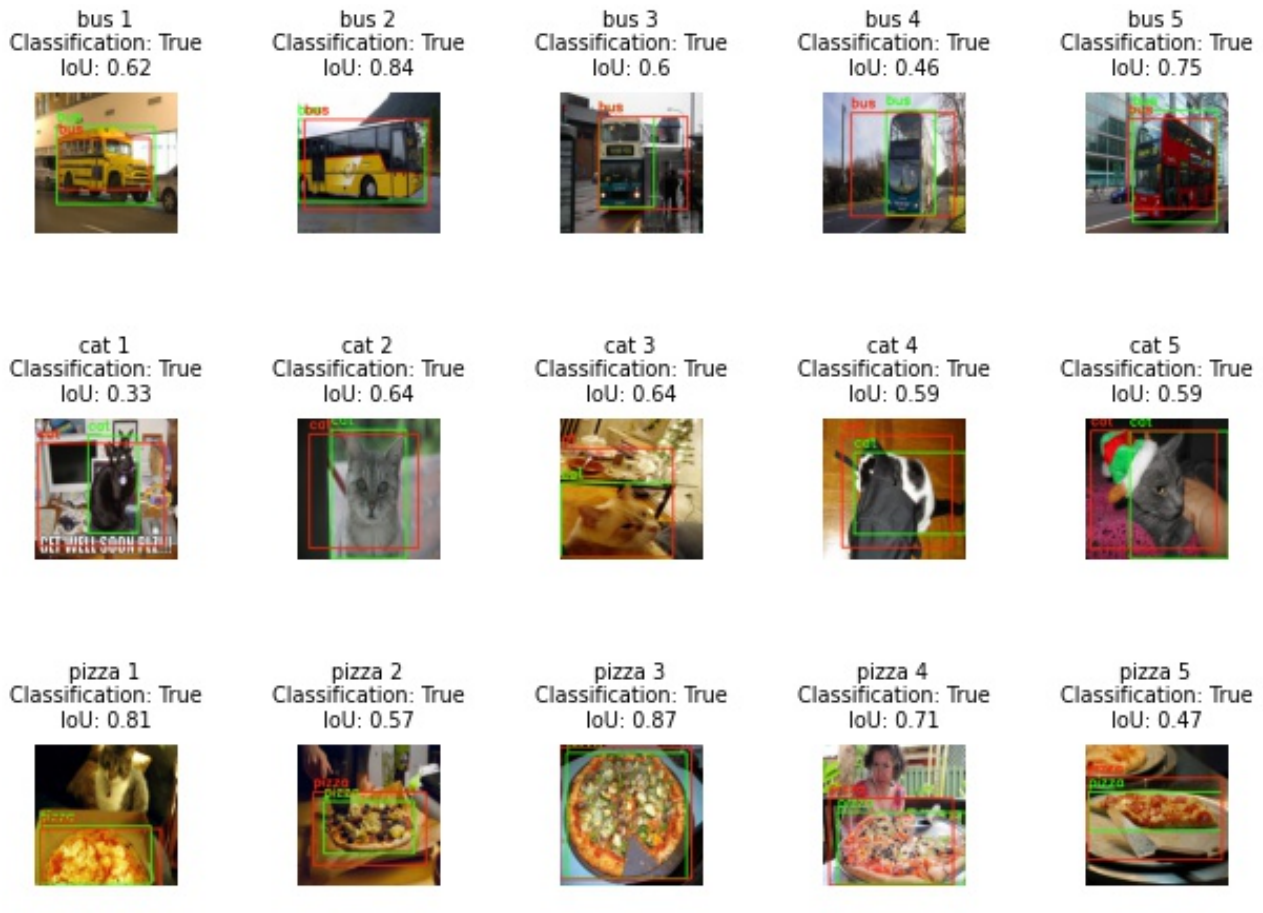



Figure 15: Sample predictions of CIoU model over **train data**

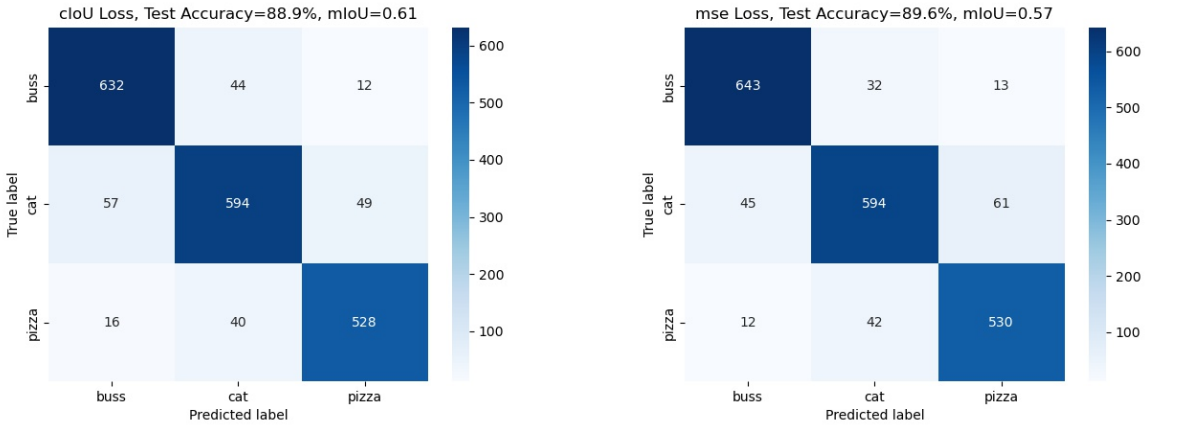
was used for all upcoming sample prediction figures. Although there are very good predictions like bus 2, pizza 3, and pizza1, there are some bad predictions as well such as bus 4, pizza 2, and pizza 5. These erroneous examples can be made better for training by increasing the number of epochs. However, as the validation loss remains constant and increased a little bit, it is not a good idea to increase the number of epochs as the model starts to memorize the samples. Lastly, one can see the label for the ground truth and predicted class at the top left of the bounding box. In Figure 15, all class predictions are correct. Next, sample predictions for the MSE model over train data can be seen in Figure 16. Its performance on train data is pretty similar to the CIoU model. That's no surprise because the CIoU model has 99.6% accuracy and 0.68 mean IoU, and mIoU model has 99.4% accuracy and 0.67 mean IoU over the train set. That implies both models are powerful enough to learn the given data.


Figure 16: Sample predictions of MSE model over **train data**

### 4.2 Test/Validation Results

In this section, I will present and discuss the results of the models on test data. Results in that section reflect the generalization performance of the models. They can be regarded as an approximate performance of an algorithm in production. Confusion matrices for two models over the test set can be seen in Figure 17. The test accuracy of the CIoU model and MSE model are 88.9% and 89.6% respectively. The mean IoU performance of the models is 0.61 and 0.57 respectively. This suggests that, as expected and mentioned in Section 3, since regression loss also affects the classification performance, CIoU, and MSE models have different classification performances. There is a 0.7% difference whereas the difference between the two models in terms of mIoU is more significant than accuracy, it is 0.04 or 4%.

From the confusion matrices, it seems that both models generalized well in terms of classification and the bounding box performances, which will be demonstrated from sample predictions. Another observation is that the models' best performance is in distinguishing the bus and pizza classes. In other words, for both models, bus and pizza classes are more distinctive than pizza and cat or bus and cat classes because the lowest misclassification was done in between bus and pizza classes.




(a) Confusion matrix of HW5Net with CIoU loss over **test data**

(b) Confusion matrix of HW5Net with MSE loss over **test data**

Figure 17: Confusion matrices for CIoU and MSE losses over **test data**

Other than these two classes, other misclassification seems distributed roughly uniformly. And obviously, most of the numbers are on the diagonal cells, which indicates the effective performance of models on classification. Another difference between CIoU and MSE models is MSE model did more misclassifications between cat and pizza whereas the ICoU model did more misclassifications between cat and bus.

Sample predictions of the CIoU model can be seen in Figure 18. Similarly, red boxes denote prediction of the model and green boxes denote the ground truth. Label of the ground truths and predictions are written at the top left corner of each bounding box. Since there is no shared sample in train and test datasets, there is no common sample between Figure 18, 19 and Figure 15, 16. One observation is the all class predictions in Figure 18 are correct. However, there are some good and bad cases for mean IoU results. For instance, bounding box prediction for bus 1 is very bad whereas it is very good for pizza 3. This demonstrates that the bounding box prediction depends on the context of the object. In bus 1, there is a crowded background, which consists of buildings and trees behind the bus. That kind of noisy and crowded context make classification more difficult. However, in pizza 3, background is not crowded and there are less noise so bounding box is much better.

Another observation is that as the size of the objects in an image decreases, bounding box performance degrades gradually. For instance, in pizza 2, predicted bounding bax is very good whereas in pizza 1, as pizza is at the small part of the image, performance decreases. It is also important that whether the portion of the object or the complete object is visible in an image. In images that give the worst IoU performance, the target object is not completely visible. As an example, in pizza 1, bus 1, bus 5, and cat 4, target objects are not complete so IoU performance is worse than other examples. Lastly, materials that affect the appearence of the target object can be treated as noisy because they affect the bounding box predictions negatively. As an example, in cat 3, ground truth box only includes the cat and excludes its hat. However, our model saw its hat as a part of the object and includes it, which prevents the model from achieving $IoU = 1$. Thus, as the number of other objects increase in the image, performance degrades. In cat 5, there is no other object besides cat, so the performance is good.


Figure 18: Sample predictions of CIoU model over **test data**

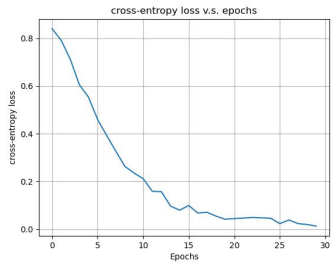
In Figure 19, sample prediction results for MSE model are presented. Similar arguments aforementioned in previous paragraphs are valid for this model too. In addition to them, robustness of the model decreases in MSE loss, which aligns with 0.04 decrease in IoU score. For example, in bus 4, the lower part of the image includes bus and other part includes distracting objects. MSE model's performance on that example is 0.04 less than CIoU model. That difference is significant in bus 5. These observations suggest that MSE model performs worse in difficult examples than CIoU model. Another observation is that in some of the easy examples, that includes all parts of the target object, MSE model performs better than CIoU model. For instance, in pizza 2, most of the image consists of the target object and MSE model performs 0.05 better than CIoU. I think the reason for that is because when there are small objects, initial loss can be very high if the prediction and the ground truth boxes are far from each other. Since MSE weights anomalies too much, this may decrease the performance. However, since CIoU considers the characteristic features of boxes such as ratio of diagonals, aspect ratios, and intersection over union, it generates more accurate boxes for difficult cases. For the easy examples or large ground truth boxes, as it is very likely to have intersections between two boxes, focusing on the corners can be better idea. Because, as we saw in the class, IoU can remain the same but there can be different MSE losses which correspond to very different predictions. Thus, it may be a good idea to use MSE criteria for such images.


Figure 19: Sample predictions of MSE model over **test data**

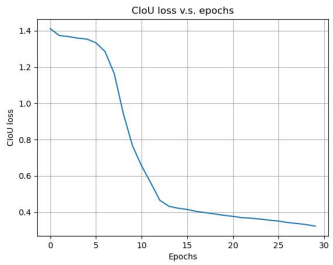
From the training results, we can infer that both CIoU and MSE models are able to learn the train data as classification accuracy is 99% and I observed mIoU approaches to 1 as I increased the number of epochs. In my first attempt to solve this homework, I implemented Resnet52 but its training time was much longer than my current architecture, and loss decrease slowly due to a large number of convolutional layers. Thus, I decided to try a simpler network to learn the data, and using the template provided in the homework, I implemented `HW5Net` with Resnet34 skip-blocks. Since I observed that this model is powerful for the given dataset, I did not make it denser and implement Resnet52.

### 4.3 Training Curves/Histories

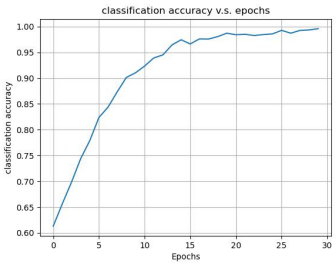
Lastly, although this is not asked in homework, I want to put my training loss and accuracy values recorded at each epoch to give the TA another proof that my network is learning. You can see them in Figure 20. Cross-entropy and CIoU get very close to 0 and classification accuracy gets very close to 100%, which are implications of the following the network is powerful enough and learn the data and there is no vanishing gradient problem thanks to the residual blocks. To assess its generalization performance, one can look at the mIoU score on test data and the confusion matrix




(a) Cross-entropy training loss at each epoch




(b) CIoU training loss at each epoch




(c) Classification accuracy at each epoch (1.0 = 100%)

Figure 20: Training losses at each epoch for **CIoU model**

on the test data, which was discussed in previous sections. Since **MSE model** has a very similar curve, I did not put it here.

### 4.4 How to improve the results?

There are various ways of improving the generalization performance of the current network. One of the problems with the current network is it is not robust to different objects, which are not defined. For instance, in Figure 18 cat 3, the hat is counted as a part of the cat. To prevent this, one may add random shapes to each image as an augmentation method. This will require more epochs to train the image but it may increase the performance. Furthermore, random rotations and random crops can be added to increase the performance for incomplete objects in images. In addition to augmentation techniques, we may increase the data size to increase the generalization performance. About the model architecture, I think it may also make sense to add skip connections at every 2 or 3 skip blocks so that gradients can be more effective at deep layers. At the beginning of the dataset preparation, we discarded a great amount of data. Instead of doing that, we may build a network for multi-object detection and use all the images and objects in the training process. Lastly, for image classification and object detection, we may implement transformer architecture. Thanks to its attention mechanism, it may result in better localization and bounding boxes.

### 4.5 How to run the code?

`hw5_MehmetBerkSahin.py` is the main code of this homework. To run it and get the answers of this homework successfully, first, you need to generate the dataset for this homework from COCO API by running the following command:

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

Then, you can run the following command to start the training two models with CIoU and MSE losses:

```
python hw5_MehmetBerkSahin.py
```

This will save the confusion matrices and sample predictions for both train and test data automatically to the current directory. And note that the "coco" folder should have subfolders with "annotations", "train2014", and "test2014", which include 2014 Train/Val annotations, 2014 Train images, and 2014 Val images respectively from COCO webpage. Lastly, training of this code was done with NVIDIA A100 GPU so the program may give an error if there is a different GPU or CPU.

# 5 Lessons Learned

In this homework, I learned the implementation of Resnet architecture with different skip-connection types (Resnet34 and Resnet52). I learned how to design my own residual block and put it into my neural network architecture. I learned how to design an object detection architecture with classification and regression heads and learned how to do backward pass through them. I played with the annotations and learned how to resize them with the images and learned how to use them in training. I learned Complete IoU and mean IoU loss functions and learned how to use MSE and IoU-type functions for bounding box prediction. I learned using `opencv` library to display the predicted bounding boxes and classification results with their ground truths. It was pretty good homework and looking forward to the next one.

# 6 Source Code

## 6.1 `network.py`

```
import torch.nn as nn
import torch.nn.functional as F
import torch
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 HW5Net(nn.Module):

    def __init__(self, in_channels=3, width=8, n_blocks=5, learnable_res=False):
        assert (n_blocks >= 0)
        super(HW5Net, self).__init__()
        # 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),
                      ]
            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, 1024),
            nn.BatchNorm1d(1024),
```

```
            nn.ReLU(True),
            nn.Linear(1024, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(True),
            nn.Linear(64, 3),
        )
        # bounding box detector head
        self.bbox_head = nn.Sequential(
            nn.Linear(32768, 5196),
            nn.BatchNorm1d(5196),
            nn.ReLU(True),
            nn.Linear(5196, 1024),
            nn.BatchNorm1d(1024),
            nn.ReLU(True),
            nn.Linear(1024, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(True),
            nn.Linear(256, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(True),
            nn.Linear(64, 4))

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

# test code
if __name__ == "__main__":
    """
    # test the Bottleneck block
    x = torch.randn((4,3,256,256))
    block1 = BottleNeck(in_channels=3, out_channels=3, width=3, downsample=False)
    y = block1(x)
    print(f"x shape: {x.shape}")
    print(f"y shape: {y.shape}")


    x = torch.randn((4, 3, 224, 224))
    resnet = ResNet50()
    y = resnet(x)
    print(f"x shape: {x.shape}")
    print(f"y shape: {y.shape}")
    print("Parameters:", len(list(resnet.parameters())))
    """
    """
    print(torch.cuda.is_available())
    x2 = torch.randn((4, 3, 256, 256))
    beast = BeastNet()
    y1, y2 = beast(x2)
    print("y1 shape:", y1.shape)
    print("y2 shape:", y2.shape)

    # test localizer
    x3 = torch.randn((4, 64, 256, 256))
    localizer = Localizer()
    y = localizer(x3)
    print("Localizer output shape:", y.shape)

    x = torch.randn((4,3,256,256))
    model = HW5Net()
    cat, bbox = model(x)
    print(cat.shape)
    print(bbox.shape)
    """
    x = torch.randn((4, 3, 256, 256))
    model = Block(3, learnable_res=True)
    y = model(x)
    print("Output shape:", y.shape)
    print("Block works sucessfully")

    x = torch.randn((4,3,256,256))
    model = HW5Net(learnable_res=False)
    cat, bbox = model(x)
    print("Whole network works sucessfully")

    # print model summary
    model = HW5Net()
    # show output shape and hierarchical view of net
    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())))
```

## 6.2 utils.py

```
import numpy as np
from torchvision.ops import box_iou
import torch.nn as nn
from torchvision.ops import complete_box_iou_loss
from network import HW5Net
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
import torch
import cv2


class mIoU(nn.Module):
    def __init__(self):
        super(mIoU, self).__init__()

    def forward(self, output, target):
        # for single sample (inference)
        if output.shape[0] == 1:
            return box_iou(output, target.unsqueeze(0))
        # for batches (training)
        else:
            mean_loss = 0
            batch_size = output.shape[0]
            # calculate and return the mean loss (1 - mIoU)
            for i in range(batch_size):
                mean_loss += (1 - box_iou(output[i].unsqueeze(0), target[i].unsqueeze(0)).item()) / batch_size
            return mean_loss

class CompleteIOULoss(nn.Module):

    def __init__(self, reduction='none'):
        super(CompleteIOULoss, self).__init__()
        self.reduction = reduction

    def forward(self, output, target):
        loss = complete_box_iou_loss(output, target, self.reduction)
        return loss

def update_cm(preds, target, cm):
    # detach the preds tensor from gpu and convert to cpu
    preds = preds.detach().cpu().numpy()
    target = target.detach().cpu().numpy()
    # update cm
    for i in range(len(preds)):
        cm[target[i], preds[i]] += 1
    return cm

def plot_cm(loss_type, cm, IoU, train_eval=False):
    # classes
    classes = ['buss', 'cat', 'pizza']
    acc = round(np.trace(cm) / np.sum(cm), 3)
    plt.figure()
    s = sns.heatmap(cm, annot=True, cmap='Blues', xticklabels=classes,
                    yticklabels=classes, fmt='g')
    s.set(xlabel='Predicted label', ylabel='True label')
    word = "Train" if train_eval else "Test"
    plt.title(f"{loss_type} Loss, {word} Accuracy={round(acc*100, 2)}%, mIoU={round(1-IoU, 2)}")
    #plt.show()
    plt.savefig(f"{word}cm_{loss_type}.jpeg")
    print("confusion matrix figure is saved...")

def display_preds(test_loader, model_name="cIoU_model", num_examples=5, train_eval=False):

    class_list = ['bus', 'cat', 'pizza']

    loss_type = model_name.split("_")[0]

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    criterion1 = nn.CrossEntropyLoss()
    criterion2 = nn.MSELoss() if "mse" in model_name else CompleteIOULoss('mean')

    model = HW5Net()
    model.load_state_dict(torch.load(model_name))
    model.to(device)

    cm, results = evaluation(model, test_loader, criterion1, criterion2, device)
    plot_cm(loss_type, cm, results["miou_loss"], train_eval)

    ce, reg, acc, miou = results["cross-entropy"], results["reg_loss"], results["accuracy"], 1-results["miou_loss"]
    # report the results
    print(f"MODEL NAME: {model_name}")
    print("*"*40)
    print(f"cross-entropy loss: {ce:.2f}")
    print(f"{loss_type} loss: {reg:.2f}")
    print(f"classification accuracy: {acc*100:.2f}%")
    print(f"mean IoU: {miou:.2f}")
    print("*"*40)
```

```
    # initialize counter and image dictionaries
    counter = {ids: 0 for ids in [0, 1, 2]}
    imgs = {ids: [] for ids in [0, 1, 2]}
    # stop criteria
    criteria = np.array([counter[i] >= num_examples for i in range(3)])
    stop = False
    # choose images
    for data in test_loader:
        label = data["label"]
        for idx, cat in enumerate(label):
            ids = cat.item()
            if counter[ids] < num_examples:
                counter[ids] += 1
                # create sample dict
                sample = {"image" : data["image"][idx],
                          "bbox" : data["bbox"][idx],
                          "label": data["label"][idx]}
                imgs[ids].append(sample)

            if np.all(criteria):
                stop = True
                break
        if stop:
            break

    model.to("cpu")
    model.eval()
    # display the predictions in a figure
    fig, axs = plt.subplots(3, num_examples)
    fig.tight_layout()

    for row, cat in enumerate(list(imgs.keys())):
        for col, data in enumerate(imgs[cat]):
            img, bbox, label = data["image"], data["bbox"], data["label"]
            pred_label, pred_bbox = model(img.unsqueeze(0))
            # iou loss
            iou = mIoU()(pred_bbox, bbox)
            # unwrap the tensors
            pred_label = torch.argmax(pred_label).item()
            label = label.item()
            bbox, pred_bbox = bbox.numpy(force=True) * 255, pred_bbox.numpy(force=True) * 255
            img = np.uint8(img.numpy(force=True) * 255)
            img = img.transpose((1, 2, 0))
            # get the box locations
            [x1, y1, x2, y2] = bbox
            [x1p, y1p, x2p, y2p] = pred_bbox[0]
            # ground truth
            img = np.ascontiguousarray(img)
            img = cv2.rectangle(img, (round(x1), round(y1)),
                                     (round(x2), round(y2)),
                                     (36, 256, 12), 2)

            img = cv2.putText(img, class_list[cat], (round(x1), round(y1 - 10)), cv2.FONT_HERSHEY_SIMPLEX,
                                                0.8, (36, 256, 12), 2)
            # prediction
            img = cv2.rectangle(img, (round(x1p), round(y1p)),
                                     (round(x2p), round(y2p)),
                                     (256, 36, 12), 2)

            img = cv2.putText(img, class_list[cat], (round(x1p), round(y1p - 10)), cv2.FONT_HERSHEY_SIMPLEX,
                                                0.8, (256, 36, 12), 2)
            # plot the image
            axs[row, col].imshow(img)
            axs[row, col].axis('off')
            axs[row, col].set_title(f"{class_list[cat]} {col + 1}\nClassification: {label == pred_label}\nIoU: {round(iou[0].item(), 2)}", size=7)

    word = "Train" if train_eval else "Test"
    plt.savefig(f"{model_name}_{word}preds.jpeg")
    print("Predictions are plotted and figure is saved...")


def evaluation(model, test_loader, criterion1, criterion2, device):

    cm = np.zeros((3,3))
    # TEST PART
    # initalize the losses with 0
    run_cross_loss = 0.0
    run_reg_loss = 0.0
    run_iou_loss = 0.0
    run_acc = 0.0
    # size of the dataset
    test_size = len(test_loader)
    model.eval() # evaluation mode
    for i, data in enumerate(test_loader):
        img, bbox, label = data['image'], data['bbox'], data['label']
        # load data to device
        img = img.to(device)
        bbox = bbox.to(device)
        label = label.to(device)

        # test
        output = model(img)
```

```
        pred_cat = output[0]
        pred_bbox = output[1]

        # calculate loss
        cross_loss = criterion1(pred_cat, label)
        reg_loss = criterion2(pred_bbox, bbox)

        # accuracy
        _, pred = torch.max(pred_cat, 1)
        acc = torch.eq(pred, label).float().mean().item()

        # calculate mIoU loss
        miou_loss = mIoU()(pred_bbox, bbox)

        # update confusion matrix
        cm = update_cm(pred, label, cm)
        # update the losses with batch mean loss
        run_cross_loss += cross_loss.item()
        run_reg_loss += reg_loss.item()
        run_iou_loss += miou_loss
        run_acc += acc

    # calculate mean evaluations
    run_cross_loss /= test_size
    run_reg_loss /= test_size
    run_iou_loss /= test_size
    run_acc /= test_size
    # return the mean losses
    return cm, {"cross-entropy": run_cross_loss,
                "reg_loss": run_reg_loss,
                "accuracy": run_acc,
                "miou_loss": run_iou_loss}
```

## 6.3 Dataset.py

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

class MyCOCODataset(nn.Module):
    def __init__(self, data, label_map, train=True):
        super(MyCOCODataset, self).__init__()
        self.files = list(data.keys())
        self.data = data
        self.label_map = label_map
        # path of extracting data
        if train:
            self.path = 'train_data'
        else:
            self.path = 'test_data'

        self.transforms = tvt.Compose([tvt.ToTensor()
                                       #tvt.RandomCrop(224,224)])
                                      ])
    def __len__(self):
        return len(self.files) # data size

    def __getitem__(self, item):
        img_file = self.files[item]
        # get the image with transformations
        img = io.imread(os.path.join(self.path, img_file))
        #img = Image.open(os.path.join(self.path, img_file))
        img = self.transforms(img)
        # get the label
        label = self.label_map[self.data[img_file]['category_id']]
        # get the bounding box
        [x1, y1, w, h] = self.data[img_file]['bbox']
        x2, y2 = x1+w, y1+h # lower right corner
        return {"image": img,
                "label": torch.tensor(label),
                "bbox": torch.tensor([x1/255, y1/255, x2/255, y2/255])}



# dataset generator for train and validation sets
def data_generator(coco, catIds, data_path, train=True):
```

```
    # keeps the file names as keys and annotations as values
    data = {}

    for cat_id in catIds:
        imgIds = coco.getImgIds(catIds=cat_id)
        for img_id in imgIds:
            # get annotations
            annIds = coco.getAnnIds(imgIds=img_id, catIds=cat_id, iscrowd=False,
                                    areaRng=[200 * 200, float('inf')])
            # load annotations
            anns = coco.loadAnns(annIds)
            # use only images with one dominant bbox
            if len(anns) != 1:
                continue
            ann = anns[0]

            # read the image and resize
            img = coco.loadImgs(img_id)[0]
            file_name = img['coco_url'].split('/')[-1]
            I = io.imread(os.path.join(data_path, file_name))
            if len(I.shape) == 2:
                I = skimage.color.gray2rgb(I)
            # retrieve image width and height
            img_h, img_w = I.shape[0], I.shape[1]
            I = resize(I, (256, 256), anti_aliasing=True, preserve_range=True)
            image = np.uint8(I) # image format [0, 255]
            # save the image
            if train:
                io.imsave(os.path.join("train_data", file_name), image)
            else:
                io.imsave(os.path.join("test_data", file_name), image)

            # scale annotations (bounding boxes)
            x_scale, y_scale = 256 / img_h, 256 / img_w
            [x, y, w, h] = ann['bbox']
            ann['bbox'] = [x * y_scale, y * x_scale, w * y_scale, h * x_scale]
            # delete unnecessary elements
            del ann['segmentation'], ann["iscrowd"]
            # save annotations with its image to a train dict.
            data[file_name] = ann

    return data

def plot_images(class_list, coco_inv_labels, data, catIds, img_nmb=5):
    # initialize counter and image dictionaries
    counter = {ids: 0 for ids in catIds}
    imgs = {ids: [] for ids in catIds}
    # choose img_nmb (5) elements from each category
    file_list = list(data.keys())
    random.seed(6)
    random.shuffle(file_list)

    for file_name in file_list:
        cat_id = data[file_name]['category_id']
        if counter[cat_id] < img_nmb:
            counter[cat_id] += 1
            imgs[cat_id].append(file_name)
    # plot them
    fig, axs = plt.subplots(3, img_nmb)
    fig.tight_layout()

    for row, cat in enumerate(list(imgs.keys())):
        for col, file_name in enumerate(imgs[cat]):
            class_name = class_list[coco_inv_labels[cat]]
            # load annotations
            [x, y, w, h] = data[file_name]['bbox']
            # load image
            I = io.imread(os.path.join("train_data", file_name))
            image = np.uint8(I)
            # draw bounding box
            image = cv2.rectangle(image, (round(x), round(y)),
                                  (round(x + w), round(y + h)),
                                  (36, 256, 12), 2)
            # plot the image
            axs[row, col].imshow(image)
            axs[row, col].axis('off')
            axs[row, col].set_title(f"{class_name} {col + 1}", size=7)
            if class_name == "pizza":
                print(file_name)

    plt.show()

# test code
if __name__ == "__main__":


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

    args = parser.parse_args()
    # important directories
```

```
    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")

    # train and test COCOs
    coco_train = COCO(os.path.join(ann_dir, train_json))
    coco_test = COCO(os.path.join(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

    # save inverse label map
    with open('inv_map.pkl', 'wb') as f:
        pickle.dump(coco_inv_labels, f)
        print("Inverse map saved.")

    print("datasets are being generated...")
    train_data = data_generator(coco_train, catIds, train_dir, train=True)
    val_data = data_generator(coco_test, catIds, test_dir, train=False)
    print("datasets were generated. ")
    plot_images(class_list, coco_inv_labels, train_data, catIds)
    print("sample images are plotted.")
    # save dictionaries to be used later
    with open('train_data.pkl', 'wb') as f:
        pickle.dump(train_data, f)
        print("train data saved successfully to file")
    with open('test_data.pkl', 'wb') as f:
        pickle.dump(val_data, f)
        print("test data saved successfully to file")
    # read train dictionary/data
    with open('train_data.pkl', 'rb') as f:
        train_data = pickle.load(f)
    # test train dataset
    data = MyCOCODataset(train_data, coco_inv_labels)
    print("Length of the dataset:", len(data))
    out = data[0]
    print("image:", out["image"].shape)
    print("label:", out["label"])
    print("bbox:", out["bbox"])
```

## 6.4 train.py

```
import argparse
from Dataset import MyCOCODataset
from network import HW5Net
from utils import update_cm, mIoU, evaluation
import pickle
from torchvision.ops import complete_box_iou_loss
import torch.nn as nn
import torch
from utils import CompleteIOULoss
from torch.utils.data import DataLoader
import random
import numpy as np

def train(net, num_epochs, batch_size, cIoU=True, model_name='best_model'):

    model = net
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    print(f"training is runing on {device}")
    # load train, test and inv. map dictionaries
    with open('train_data.pkl', 'rb') as f:
        train_dict = pickle.load(f)
    with open('test_data.pkl', 'rb') as f:
        test_dict = pickle.load(f)
    with open('inv_map.pkl', 'rb') as f:
        inv_map = pickle.load(f)
    print("train, test, and inv. map are loaded.")

    # train and test dataset
    train_data = MyCOCODataset(train_dict, inv_map, train=True)
    test_data = MyCOCODataset(test_dict, inv_map, train=False)
    # train and test dataloaders
    train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=True)
    train_size = len(train_loader)
    # train parameters
```

```
criterion1 = nn.CrossEntropyLoss()
if cIoU:
    criterion2 = CompleteIOULoss('mean')
    loss_name = "CIoU"
else:
    criterion2 = nn.MSELoss()
    loss_name = "MSE"
# choose Adam optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, betas=(0.9, 0.99))
# train history
train_history = {
    "cross_loss": [],
    "bbox_loss" : [],
    "acc" : []
}
test_history = {
    "cross_loss": [],
    "bbox_loss": [],
    "acc": []
}

max_model_score = 0
print(f"training for {model_name} loss is in progress...")
for epoch in range(1, num_epochs+1):
    # TRAIN PART
    run_cross_loss = 0.0
    run_reg_loss = 0.0
    run_iou_loss = 0.0
    run_acc = 0.0
    model.train()
    for i, data in enumerate(train_loader):
        img, bbox, label = data['image'], data['bbox'], data['label']
        # load data to device
        img = img.to(device)
        bbox = bbox.to(device)
        label = label.to(device)

        # start train
        optimizer.zero_grad()
        output = model(img)
        pred_cat = output[0]
        pred_bbox = output[1]

        # calculate loss
        cross_loss = criterion1(pred_cat, label)
        cross_loss.backward(retain_graph=True)
        reg_loss = criterion2(pred_bbox, bbox)
        reg_loss.backward()
        optimizer.step()

        # accuracy
        _, pred = torch.max(pred_cat, 1)
        acc = torch.eq(pred, label).float().mean().item()

        # mIoU loss
        miou_loss = mIoU()(pred_bbox, bbox)

        run_cross_loss += cross_loss.item()
        run_reg_loss += reg_loss.item()
        run_iou_loss += miou_loss
        run_acc += acc
        #print(f"[Iteration {i+1}/{train_size}]")

    # calculate mean evaluations
    run_cross_loss /= train_size
    run_reg_loss /= train_size
    run_iou_loss /= train_size
    run_acc /= train_size
    # report results
    print("*"*12, "TRAIN", "*"*12)
    print(f"[epoch {epoch}/{num_epochs}] train cross entropy loss: {round(run_cross_loss,4)}")
    print(f"[epoch {epoch}/{num_epochs}] train {loss_name}: {round(run_reg_loss, 4)}")
    print(f"[epoch {epoch}/{num_epochs}] train mIoU loss: {round(run_iou_loss, 4)}")
    print(f"[epoch {epoch}/{num_epochs}] train classification accuracy {round(run_acc*100, 2)}")
    print("*"*30)
    # save the losses
    train_history["cross_loss"].append(run_cross_loss)
    train_history["bbox_loss"].append(run_reg_loss)
    train_history["acc"].append(run_acc)

    # TEST PART
    _, results = evaluation(model, test_loader, criterion1, criterion2, device)
    # get the results
    run_cross_loss = results["cross-entropy"]
    run_reg_loss = results["reg_loss"]
    run_iou_loss = results["miou_loss"]
    run_acc = results["accuracy"]
    # report results
    print("*"*12, "TEST", "*"*12)
    print(f"[epoch {epoch}/{num_epochs}] test cross entropy loss: {round(run_cross_loss, 4)}")
    print(f"[epoch {epoch}/{num_epochs}] test {loss_name}: {round(run_reg_loss, 4)}")
    print(f"[epoch {epoch}/{num_epochs}] test mIoU loss: {round(run_iou_loss, 4)}")
```

```
        print(f"[epoch {epoch}/{num_epochs}] test classification accuracy {round(run_acc*100, 2)}")
        print("*"*30)
        # save the losses
        test_history["cross_loss"].append(run_cross_loss)
        test_history["bbox_loss"].append(run_reg_loss)
        test_history["acc"].append(run_acc)

        model_score = round(run_acc, 3) + (1 - run_reg_loss)
        if model_score > max_model_score:
            torch.save(model.state_dict(), model_name)
            print("best model is saved...")
            max_model_score = model_score


    return {"model" : model,
            "train_history" : train_history,
            "test_history" : test_history}

if __name__ == '__main__':

    seed = 0
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    numpy.random.seed(seed)
    torch.backends.cudnn.deterministic=True
    torch.backends.cudnn.benchmarks=False


    parser = argparse.ArgumentParser()
    parser.add_argument("--epoch", default=15, type=int, help="numbere of epochs for training")
    parser.add_argument("--batch_size", default=16, type=int, help="batch size")

    args = parser.parse_args()

    EPOCH = args.epoch
    BATCH_SIZE = args.batch_size

    #model = BeastNet()
    model = HW5Net()
    results = train(model, EPOCH, BATCH_SIZE)

    with open('results.pkl', 'wb') as f:
        pickle.dump(results, f)
        print("results are saved.")
```

## 6.5 hw5_MehmetBerkSahin.py

```
import argparse
from Dataset import MyCOCODataset
from network import HW5Net
from train import train, evaluation
import pickle
import torch.nn as nn
import torch
from torch.utils.data import DataLoader
import random
import numpy
from utils import display_preds


if __name__ == "__main__":
    # for reproducible results
    seed = 0
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    numpy.random.seed(seed)
    torch.backends.cudnn.deterministic=True
    torch.backends.cudnn.benchmarks=False
    # take the user defined variables
    parser = argparse.ArgumentParser()
    parser.add_argument("--epoch", default=15, type=int, help="numbere of epochs for training")
    parser.add_argument("--batch_size", default=16, type=int, help="batch size")

    args = parser.parse_args()

    EPOCH = args.epoch
    BATCH_SIZE = args.batch_size

    # TRAININGN PART
    # train HW5Net with CIoU
    model_cIoU = HW5Net()
    exp1 = train(model_cIoU, EPOCH, BATCH_SIZE, True, 'cIoU_modelR')
```

```
# train HW5Net with MSE
model_mse = HW5Net()
exp2 = train(model_mse, EPOCH, BATCH_SIZE, False, 'mse_model')

# save the results
with open('cIoU_model_results.pkl', 'wb') as f:
    pickle.dump(exp1, f)
with open('mse_model_results.pkl', 'wb') as f:
    pickle.dump(exp2, f)
print("experiment results were saved.")

# load train data, test data and label mapping
with open('train_data.pkl', 'rb') as f:
    train_dict = pickle.load(f)
with open('test_data.pkl', 'rb') as f:
    test_dict = pickle.load(f)
with open('inv_map.pkl', 'rb') as f:
    inv_map = pickle.load(f)

# EVALUATION PART
# evaluate on test set
test_data = MyCOCODataset(test_dict, inv_map, train=False)
test_loader = DataLoader(test_data, batch_size=BATCH_SIZE, shuffle=False)

# display predictions of model trained with cIoU and mse model (over test data)
display_preds(test_loader, "cIoU_model")
display_preds(test_loader, "mse_model")

# evaluate on train set
train_data = MyCOCODataset(train_dict, inv_map, train=True)
train_loader = DataLoader(train_data, batch_size=BATCH_SIZE, shuffle=False)

# display predictions of model trained with cIoU and mse model (over train data)
display_preds(train_loader, "cIoU_model", train_eval=True)
display_preds(train_loader, "mse_model", train_eval=True)
```

# References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.