BBM418 Computer Vision Lab.
Assignment 4 - Object Tracking on Videos

Fatma Nur Demirbaş
21727116
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
b21727116@cs.hacettepe.edu.tr

Overview

In this assignment, I implemented and practice with object recognition. This assignment has two parts. In Part 1, I implemented the basic object recognition+localization pipeline. In Part 2, I used YOLOv3 object detection with mean-shift algorithm for tracking.

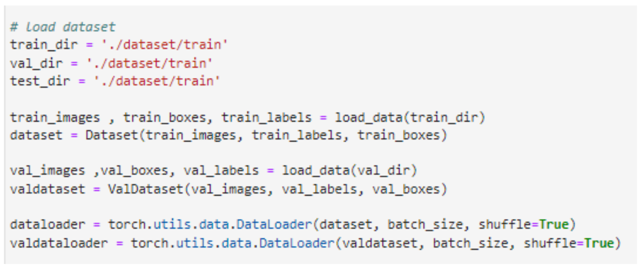
1 PART 1

1.1 Introduction

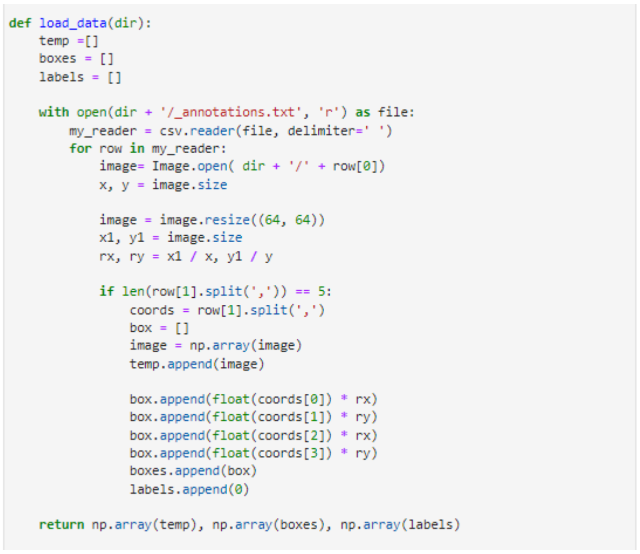
In this section, the Raccoon dataset was used to localize raccoons using the classification and localization framework and detection raccoon is done.

1.2 Implementation Details

Dataset Splitting: Visualize the dataset with bounding boxes before moving on to the machine learning section of this lesson. By multiplying the image size, we can find the coordinates. To display the image, we are using OpenCV. We now need to partition the dataset before moving on to DataLoaders. We start creating new PyTorch DataLoaders for our dataset, which is dispersed among variables, when the dataset split is completed.

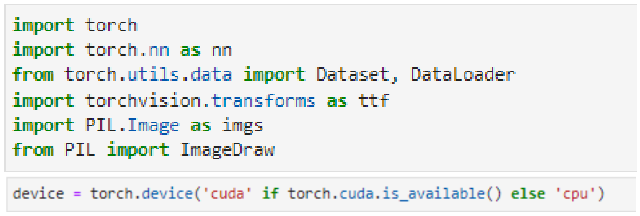


```
# Load dataset
train_dir = './dataset/train'
val_dir = './dataset/train'
test_dir = './dataset/train'

train_images , train_boxes, train_labels = load_data(train_dir)
dataset = Dataset(train_images, train_labels, train_boxes)

val_images ,val_boxes, val_labels = load_data(val_dir)
valdataset = ValDataset(val_images, val_labels, val_boxes)

dataloader = torch.utils.data.DataLoader(dataset, batch_size, shuffle=True)
valdataloader = torch.utils.data.DataLoader(valdataset, batch_size, shuffle=True)
```



```
def load_data(dir):
    temp =[]
    boxes = []
    labels = []

    with open(dir + '/_annotations.txt', 'r') as file:
        my_reader = csv.reader(file, delimiter=' ')
        for row in my_reader:
            image= Image.open( dir + '/' + row[0])
            x, y = image.size

            image = image.resize((64, 64))
            x1, y1 = image.size
            rx, ry = x1 / x, y1 / y

            if len(row[1].split(',')) == 5:
                coords = row[1].split(',')
                box = []
                image = np.array(image)
                temp.append(image)

                box.append(float(coords[0]) * rx)
                box.append(float(coords[1]) * ry)
                box.append(float(coords[2]) * rx)
                box.append(float(coords[3]) * ry)
                boxes.append(box)
                labels.append(0)

    return np.array(temp), np.array(boxes), np.array(labels)
```

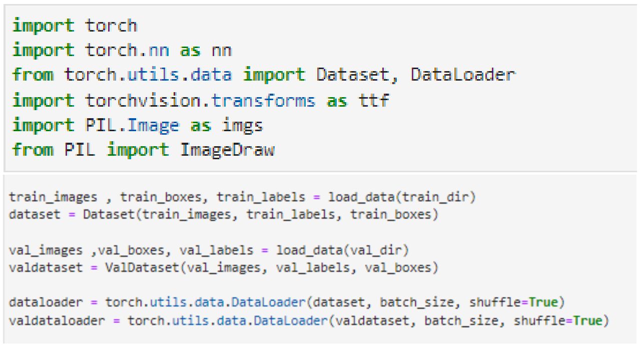
Custom DataLoaders in Pytorch DataLoaders produces an object that will handle the whole data supply system while the model is being trained, as the name implies. It has features like shuffle as you create the object and a 'getitem' method that manages what your data entry should be at each iteration, allowing you to design everything the way you want without making the code in the tutorial part messy. This frees up time to focus on other improvements. Using GPUs to train ML models is one of the most crucial things to accomplish, especially when the target is large. A system with GPU should be chosen if you are working in Paperspace Gradient.This indicates that it is a GPU and the device can be used to convert data and models to GPU for exploitation.



```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as ttf
import PIL.Image as imgs
from PIL import ImageDraw
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

We'll start by loading the image, labels, and box coordinates that scale from 0 to 1 into the Dataset class variables. Then, at each iteration, we use 'getitem' to construct the installer output. Similarly, the ValDataset (validation dataset) DataLoader class will be created. We shall inherit from the aforementioned class because the data structure and nature are the same.



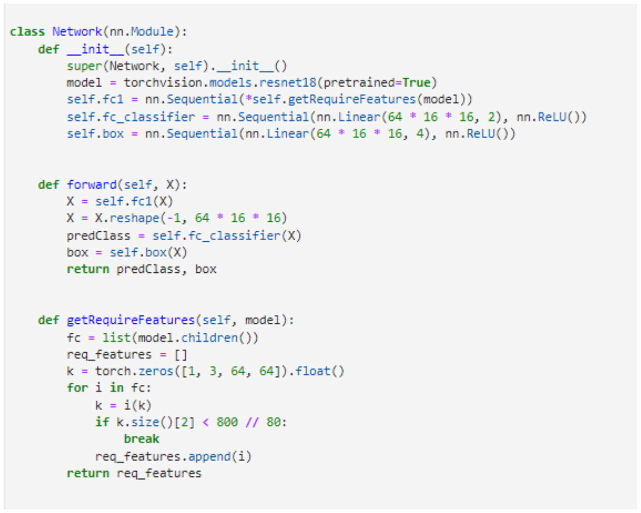
```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as ttf
import PIL.Image as imgs
from PIL import ImageDraw
```

```
train_images , train_boxes, train_labels = load_data(train_dir)
dataset = Dataset(train_images, train_labels, train_boxes)

val_images ,val_boxes, val_labels = load_data(val_dir)
valdataset = ValDataset(val_images, val_labels, val_boxes)

dataloader = torch.utils.data.DataLoader(dataset, batch_size, shuffle=True)
valdataloader = torch.utils.data.DataLoader(valdataset, batch_size, shuffle=True)
```

Now that we've got the DataLoaders Classes, we need to make data loader objects out of them. Machine Learning model of the queue tutorial has its data ready. Object localization has been achieved using a relatively simple set of Convolutional Neural Networks, which will aid in our comprehension of the idea.

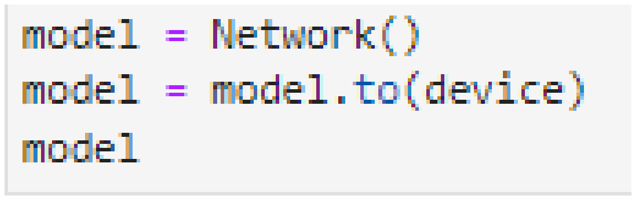
Model Architecture To comprehend how architecture should be designed, we must first comprehend the inputs and outputs. Because the input is a collection of photos, it will be styled (BS, C, H, W). Lot Size comes first, then Channels, Height, and Weight. The sequence is significant because this is how photos are stored in PyTorch. For each image in TensorFlow, it's (H, W, C).

In terms of outputs, we have two, as we described at the start of the previous blog. The first is your classification output, which will have a size of (1, N), where N denotes the number of classes. The second output is size (1, 4), which is a range of xmin, ymin, xmax, and ymax (0,1).This will make it easier to scale the coordinates to the image size afterwards. As a result, the output will be BOX, as well as the initial classification and other coordinates.

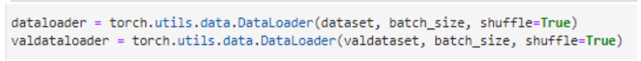


```python
class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()
        model = torchvision.models.resnet18(pretrained=True)
        self.fc1 = nn.Sequential(*self.getRequireFeatures(model))
        self.fc_classifier = nn.Sequential(nn.Linear(64 * 16 * 16, 2), nn.ReLU())
        self.box = nn.Sequential(nn.Linear(64 * 16 * 16, 4), nn.ReLU())


    def forward(self, X):
        X = self.fc1(X)
        X = X.reshape(-1, 64 * 16 * 16)
        predClass = self.fc_classifier(X)
        box = self.box(X)
        return predClass, box


    def getRequireFeatures(self, model):
        fc = list(model.children())
        req_features = []
        k = torch.zeros([1, 3, 64, 64]).float()
        for i in fc:
            k = i(k)
            if k.size()[2] < 800 // 80:
                break
            req_features.append(i)
        return req_features
```

Let's create the model and employ GPUs if they're available. This can significantly accelerate the training process, particularly for a large task like picture localisation.To feed sets of photos during training, data loaders should be constructed for both training and validation datasets.



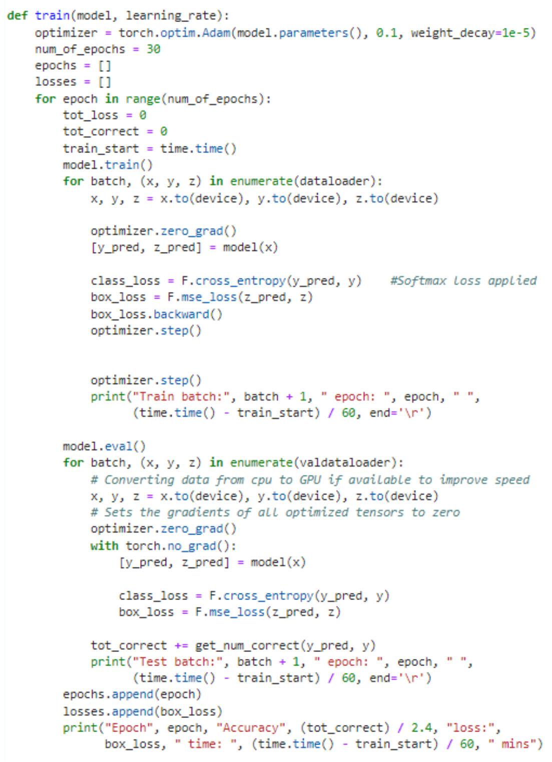
```python
model = Network()
model = model.to(device)
model
```



```python
dataloader = torch.utils.data.DataLoader(dataset, batch_size, shuffle=True)
valdataloader = torch.utils.data.DataLoader(valdataset, batch_size, shuffle=True)
```

Model Training and Validation To comprehend how architecture should be designed, we must first comprehend the inputs and outputs. Because the input is a collection of photos, it will be styled (BS, C, H, W). Lot Size comes first, then Channels, Height, and Weight. The sequence is significant because this is how photos are stored in PyTorch. For each image in TensorFlow, it's (H, W, C).

In terms of outputs, we have two, as we described at the start of the previous blog. The first is your classification output, which will have a size of (1, N), where N denotes the number of classes. The second output is size (1, 4), which is a range of xmin, ymin, xmax, and ymax (0,1).This will make it easier to scale the coordinates to the image size afterwards. As a result, the output will be BOX, as well as the initial classification and other coordinates.

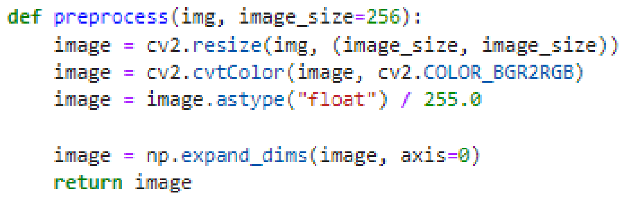


```
def train(model, learning_rate):
    optimizer = torch.optim.Adam(model.parameters(), 0.1, weight_decay=1e-5)
    num_of_epochs = 30
    epochs = []
    losses = []
    for epoch in range(num_of_epochs):
        tot_loss = 0
        tot_correct = 0
        train_start = time.time()
        model.train()
        for batch, (x, y, z) in enumerate(dataloader):
            x, y, z = x.to(device), y.to(device), z.to(device)

            optimizer.zero_grad()
            [y_pred, z_pred] = model(x)

            class_loss = F.cross_entropy(y_pred, y)    #Softmax loss applied
            box_loss = F.mse_loss(z_pred, z)
            box_loss.backward()
            optimizer.step()


            optimizer.step()
            print("Train batch:", batch + 1, " epoch: ", epoch, " ",
                  (time.time() - train_start) / 60, end='\r')

        model.eval()
        for batch, (x, y, z) in enumerate(valdataloader):
            # Converting data from cpu to GPU if available to improve speed
            x, y, z = x.to(device), y.to(device), z.to(device)
            # Sets the gradients of all optimized tensors to zero
            optimizer.zero_grad()
            with torch.no_grad():
                [y_pred, z_pred] = model(x)

                class_loss = F.cross_entropy(y_pred, y)
                box_loss = F.mse_loss(z_pred, z)

            tot_correct += get_num_correct(y_pred, y)
            print("Test batch:", batch + 1, " epoch: ", epoch, " ",
                  (time.time() - train_start) / 60, end='\r')
        epochs.append(epoch)
        losses.append(box_loss)
        print("Epoch", epoch, "Accuracy", (tot_correct) / 2.4, "loss:",
              box_loss, " time: ", (time.time() - train_start) / 60, " mins")
```

The data loaders we constructed are used to access each data group, which consists of photos, labels, and bounding boxes x, y, and z, respectively. Then each gets changed to our preferred device, such as a GPU if one is available. In Deep Learning, optimizers manage backpropagation, thus we use optimizer.zero grad to set the gradients to zero for each batch before training (). The loss calculation begins once the input(x) is fed into the model. This is a crucial step since there are two sorts of losses: the **cross_entropy**() loss for classification problems (softmax loss applied).

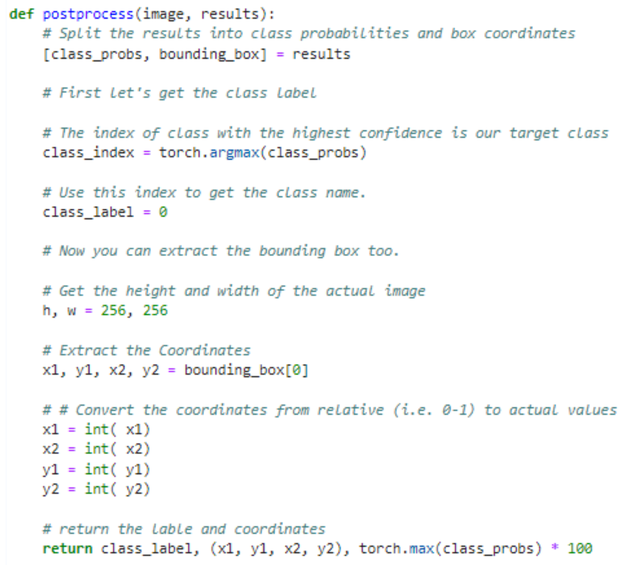
Model Testing Prediction isn't only about getting a printout from the model when it comes to image localisation. To visualize the result, we'll need to process the bounding box coordinates to build a genuine bounding box, which will be useful even in production. Prediction Script, Pre-Processing, and Post-Processing functions will be the three key components of this part.

- **Pre-Processing** It's critical when working on a project that the data you feed into the model is preprocessed in the same way that the data you enter into the training is. In subtraction, resizing is one of the most common preprocessing steps for any image-related model. Our image is 256 pixels wide.



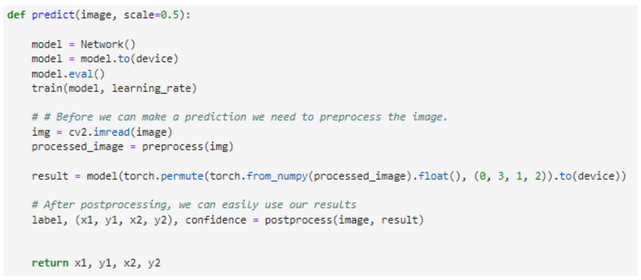
```
def preprocess(img, image_size=256):
    image = cv2.resize(img, (image_size, image_size))
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    image = image.astype("float") / 255.0

    image = np.expand_dims(image, axis=0)
    return image
```

- **Post-Processing** When we receive the printouts, they will be [clf,box], and we will need to deal with the bounding box values to generate the visualization results. Because we employ the sigmoid activation function at the end, the bounding box inputs are scaled to the [0,1] range, and our estimates are also in the [0,1] range. To exit, we'll have to rescale xmin, ymin, and so on. To do so, simply multiply the values by the image size (here 256).



```
def postprocess(image, results):
    # Split the results into class probabilities and box coordinates
    [class_probs, bounding_box] = results

    # First let's get the class label

    # The index of class with the highest confidence is our target class
    class_index = torch.argmax(class_probs)

    # Use this index to get the class name.
    class_label = 0

    # Now you can extract the bounding box too.

    # Get the height and width of the actual image
    h, w = 256, 256

    # Extract the Coordinates
    x1, y1, x2, y2 = bounding_box[0]

    # # Convert the coordinates from relative (i.e. 0-1) to actual values
    x1 = int( x1)
    x2 = int( x2)
    y1 = int( y1)
    y2 = int( y2)

    # return the lable and coordinates
    return class_label, (x1, y1, x2, y2), torch.max(class_probs) * 100
```

- **Predict** The model architecture is first retrieved from the previous network in the prediction script, and then the model is moved to the selected device. The model is set to the evaluation state, and the image is prepared for feeding into the model using the preprocessing function. The permute function in PyTorch can then be used to reconstruct the picture array from [N,H,W,C] to [N,C,H,W]. The final render function receives the result and returns the actual coordinates and label.



```
def predict(image, scale=0.5):

    model = Network()
    model = model.to(device)
    model.eval()
    train(model, learning_rate)

    # # Before we can make a prediction we need to preprocess the image.
    img = cv2.imread(image)
    processed_image = preprocess(img)

    result = model(torch.permute(torch.from_numpy(processed_image).float(), (0, 3, 1, 2)).to(device))

    # After postprocessing, we can easily use our results
    label, (x1, y1, x2, y2), confidence = postprocess(image, result)


    return x1, y1, x2, y2
```

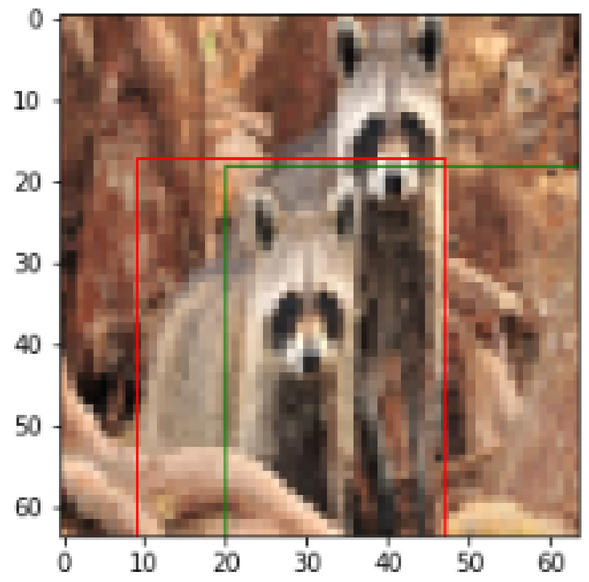
1.3 Experimental Results

Many factors influence verification accuracy, and you may not receive a favorable result on the first try. To get a true test of your model, you'll need a more extensive and generic dataset than the one I provided for testing the application. Data augmentation approaches, for example, are another option to improve the model.

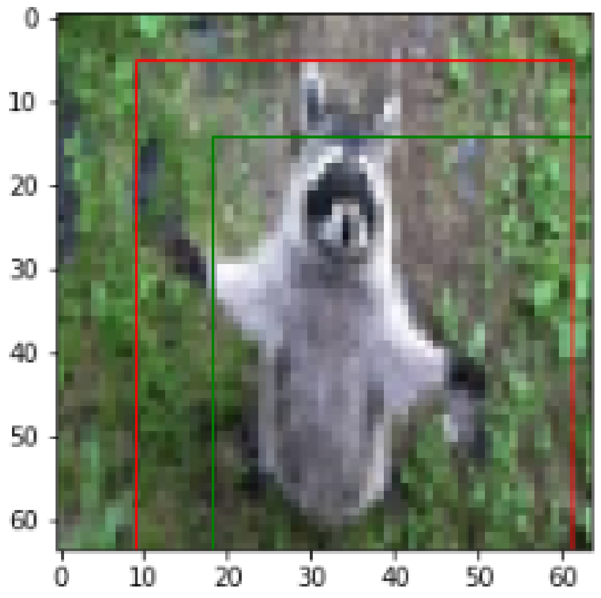
Image localization is a fascinating subject of study, and it's a little more challenging to achieve accuracy than image classification. We must not only get the categorization correct, but also a tight and accurate bounding box. It's difficult for the model to handle both, therefore increasing the dataset and architecture will assist. I hope you found this two-part post on picture localization implementation useful, and I invite you to learn more about the topic.

Best model: LR:0.01 EPOCH:50

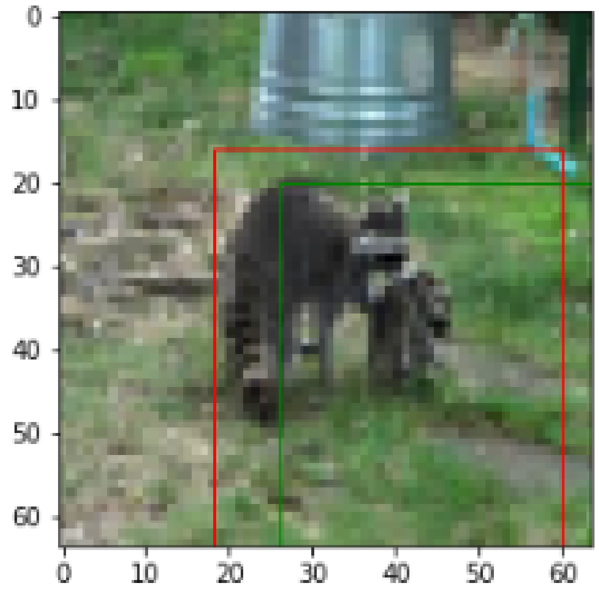
Original Bounding Boxes (red) / Predicted Bounding Boxes (green)



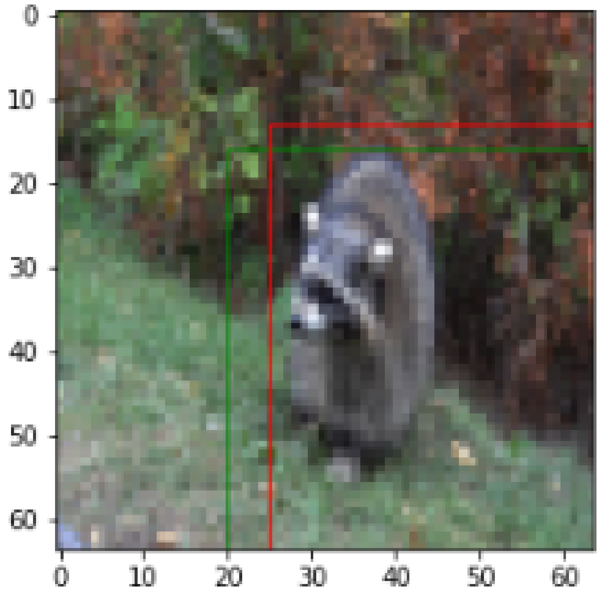
Accuracy:73.45 IOU:0.82



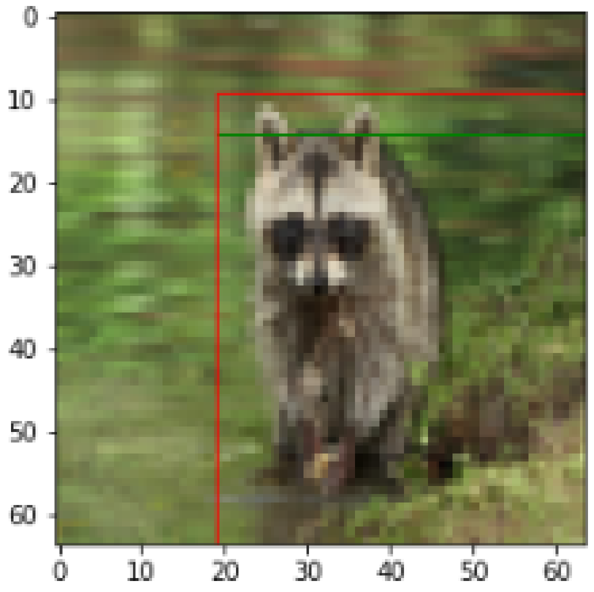
Accuracy:63.71 IOU:0.76



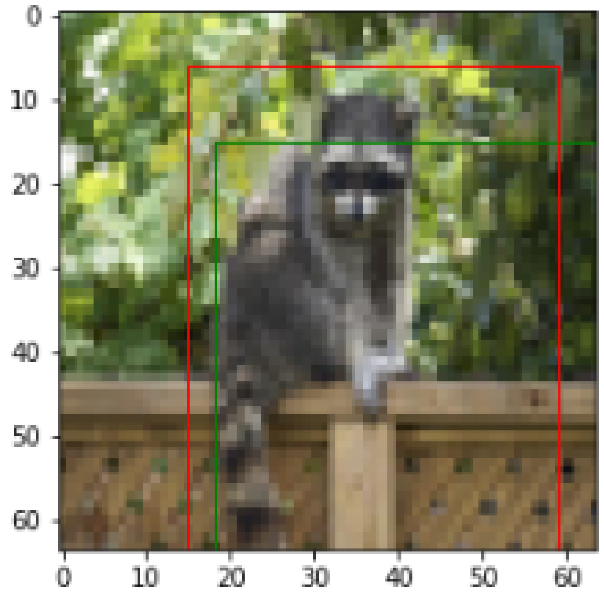
Accuracy:86.52 IOU:0.86



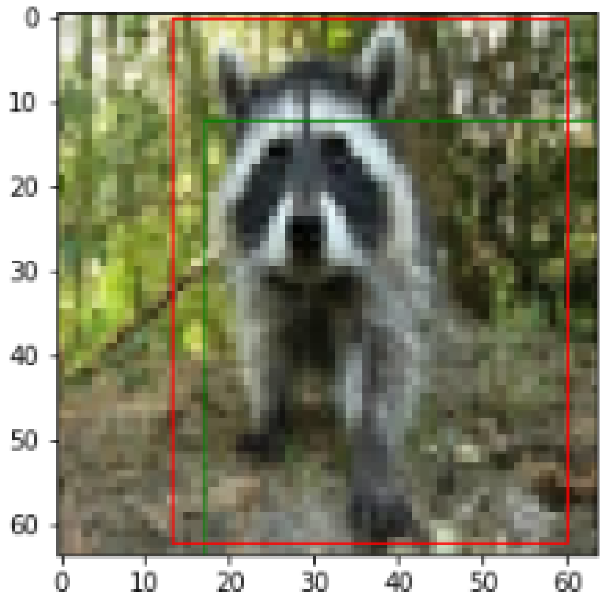
Accuracy:89.70 IOU:0.91



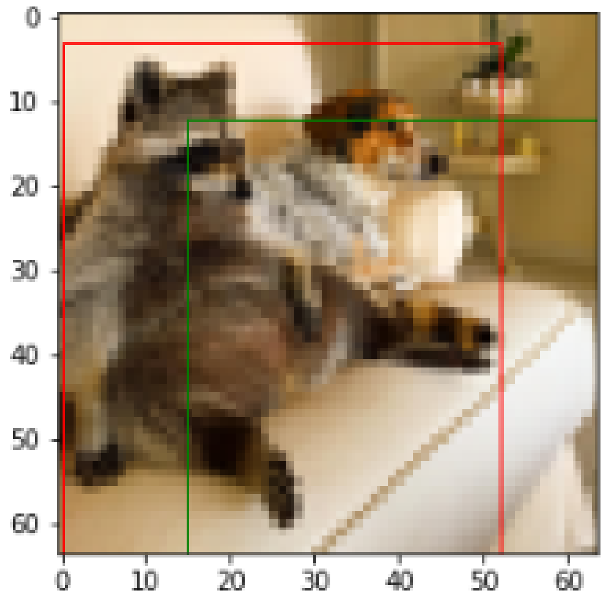
Accuracy:90.45 IOU:0.95



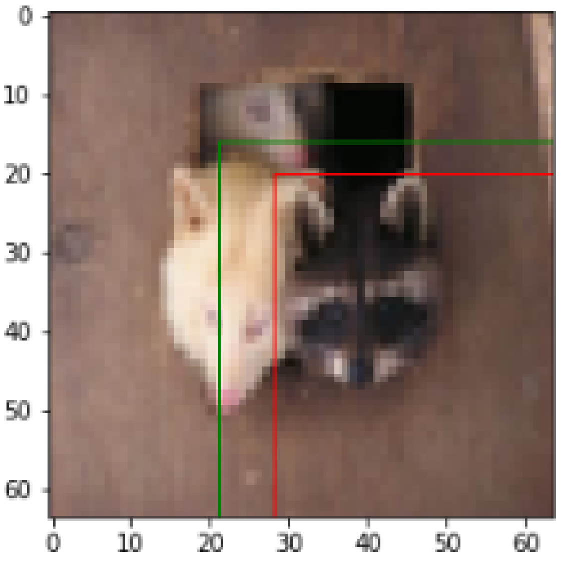
Accuracy:73.45 IOU:0.82



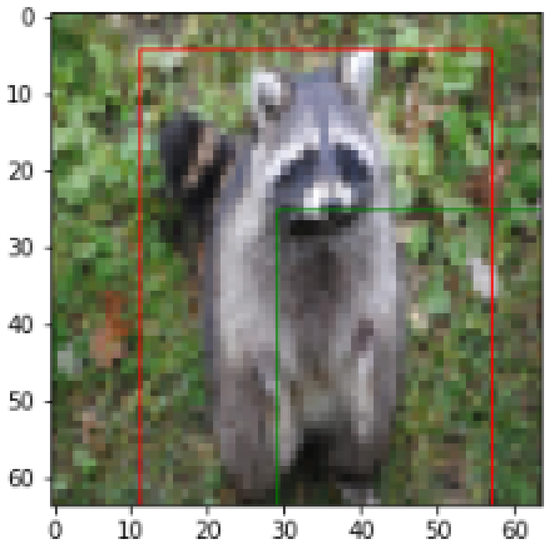
Accuracy:71.05 IOU:0.78



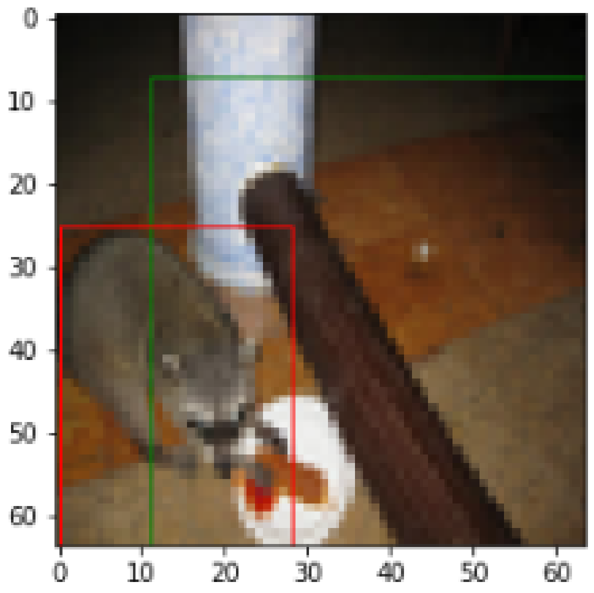
Accuracy:65.91 IOU:0.71



Accuracy:57.45 IOU:0.63



Accuracy:57.45 IOU:0.63



Accuracy:55.45 IOU:0.48

Although not very successful results can be obtained, we can say that it is a successful recognition. The applied dataset can be more maximal for full success and we need to have a more detailed and general dataset to get a real test of your model. There may have been an error in boxing and loading the data. It may be caused by the noise of the image. It is possible to train with more epochs.

2 PART 2

2.1 Introduction

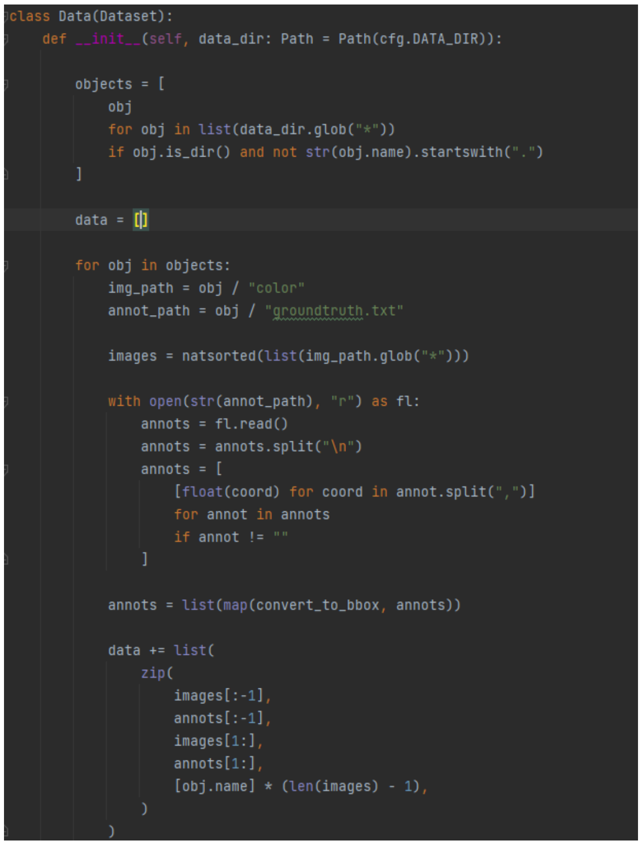
Single object tracking is image regression in which the algorithm for an image must detect a bounding box containing the object of interest. In the second approach, two images are given as input to the model. The first input is an object or a bee etc. It defines the region of interest for detection, such that in this application only a clipping of the previous frame that preserves the region of interest is taken as input. The other input is the image in which the object in the region of interest will be detected. The model detects the object by providing four values that define the coordinates of the upper-left and lower-right corners of the bounding boxes that surround the object. In the next sections, we will focus on data collection and structure and the implementation details of the model itself.

2.2 Implementation Details

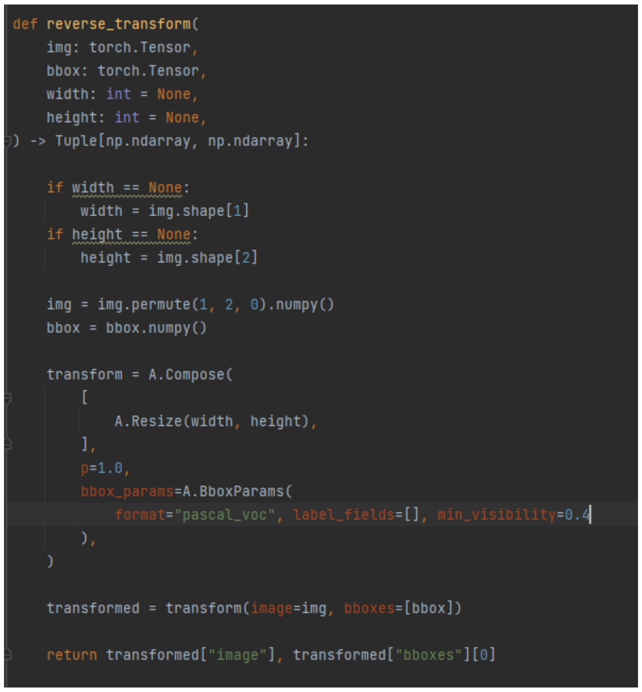
The data used for this task comes from the VOT query. Primarily VOT 2017 data. The data consists of multiple clips divided into a long frame with a bounding box description for the object of interest in each clip. There are 60 clips in total. Each clip is 100200 frames. The problem with the data is that it consists of bounding boxes of two different shapes: a polygon, x1, y1, x2, y2, ... a shape with an arbitrary number of points. A rectangle with four points x1, y1, w, h showing the upper left corner, width, and height of the bounding box. Also, the appropriate notation for the problem to be solved is different from each of the above symbols.

Aside from that, the notation that would be appropriate for the situation at hand differs from the others. x1, y1, x2, y2 is the notation we use. The rationale for this notation is that all of the values are inside the same range, i.e., 0 x w, where w is the image's width, and 0 y h, where h is the image's height. As a result, all of the values can be in the same range, and a ranging function on the model's outputs can be employed. We immediately utilize the VOTToolkit given with the challenge, which is a python API, to gather the data.

We also use a helper function that takes the output of the model with the frame entered into the model as input and returns the resized image and the corresponding bounding box as output. It basically uses scrapbook transforms to manipulate images and bounding boxes. According to the manual, the model uses the ResNet18 base for both the image input, i.e. the previous frame showing the area of interest, and the current frame. Both ResNet18 are pre-trained on ImageNet, and therefore corners, shadows, etc. They were very good at detecting features in images, eg Features from both images were extracted using ResNet18 base and then the features were combined to form a single structure. Introduction for fully connected dense layers. Both ResNet18 architectures have been removed as their output will be merged. The output of dense layers is a dimensional tensor representing the coordinates of the bounding box, i.e. (x1, y1, x2, y2).

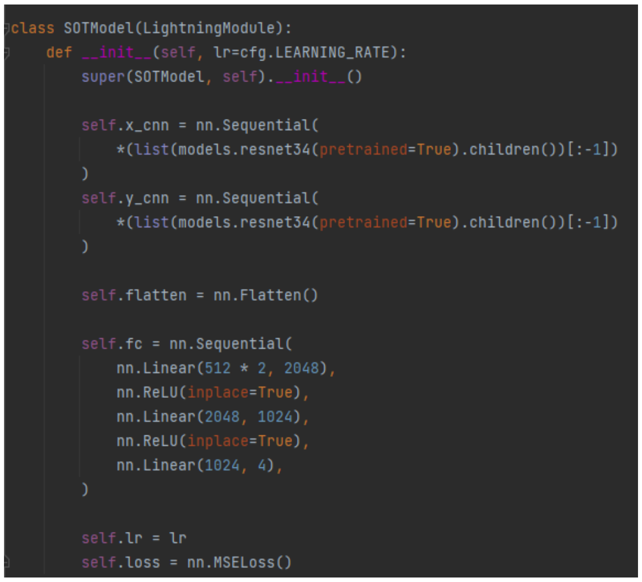


```python
class Data(Dataset):
    def __init__(self, data_dir: Path = Path(cfg.DATA_DIR)):

        objects = [
            obj
            for obj in list(data_dir.glob("*"))
            if obj.is_dir() and not str(obj.name).startswith(".")
        ]

        data = []

        for obj in objects:
            img_path = obj / "color"
            annot_path = obj / "groundtruth.txt"

            images = natsorted(list(img_path.glob("*")))

            with open(str(annot_path), "r") as fl:
                annots = fl.read()
                annots = annots.split("\n")
                annots = [
                    [float(coord) for coord in annot.split(",")]
                    for annot in annots
                    if annot != ""
                ]

            annots = list(map(convert_to_bbox, annots))

            data += list(
                zip(
                    images[:-1],
                    annots[:-1],
                    images[1:],
                    annots[1:],
                    [obj.name] * (len(images) - 1),
                )
            )
```



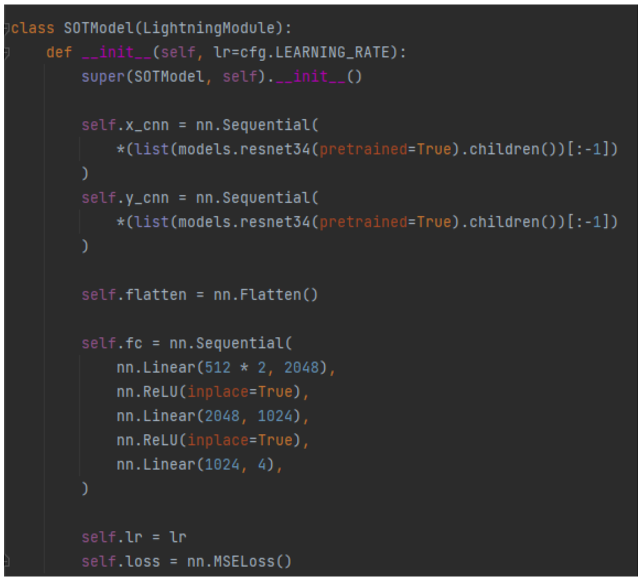
```python
def reverse_transform(
    img: torch.Tensor,
    bbox: torch.Tensor,
    width: int = None,
    height: int = None,
) -> Tuple[np.ndarray, np.ndarray]:

    if width == None:
        width = img.shape[1]
    if height == None:
        height = img.shape[2]

    img = img.permute(1, 2, 0).numpy()
    bbox = bbox.numpy()

    transform = A.Compose(
        [
            A.Resize(width, height),
        ],
        p=1.0,
        bbox_params=A.BboxParams(
            format="pascal_voc", label_fields=[], min_visibility=0.4
        ),
    )

    transformed = transform(image=img, bboxes=[bbox])

    return transformed["image"], transformed["bboxes"][0]
```

Then the data of the above structure must be loaded into memory for further processing and use. We use a custom dataset that loads the path of all images with bounding box information. It performs two transformations on the input data, first the bounding box is changed from the original format to the one we will use, and second the image and bounding box are resized to standard size 22 . The data returned by the data game during indexing is limiting. box, current frame and previous frame are truncated using the bounding box corresponding to the current frame.



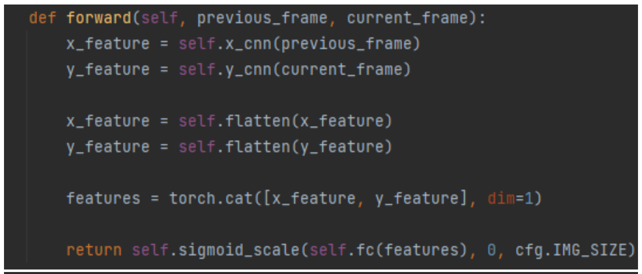
```python
class SOTModel(LightningModule):
    def __init__(self, lr=cfg.LEARNING_RATE):
        super(SOTModel, self).__init__()

        self.x_cnn = nn.Sequential(
            *(list(models.resnet34(pretrained=True).children())[:-1])
        )
        self.y_cnn = nn.Sequential(
            *(list(models.resnet34(pretrained=True).children())[:-1])
        )

        self.flatten = nn.Flatten()

        self.fc = nn.Sequential(
            nn.Linear(512 * 2, 2048),
            nn.ReLU(inplace=True),
            nn.Linear(2048, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 4),
        )

        self.lr = lr
        self.loss = nn.MSELoss()
```

The only issue with the model version is its scope. The output has a lower range. But as mentioned earlier, the request range is 0 etlt; x and lt; w and 0 etlt; y and lt; is h. So to keep the values in this range we use a simple trick of moving the values in the range (0, 1) and scaling in the range (0, w/h). So the forward method for the model is as follows:

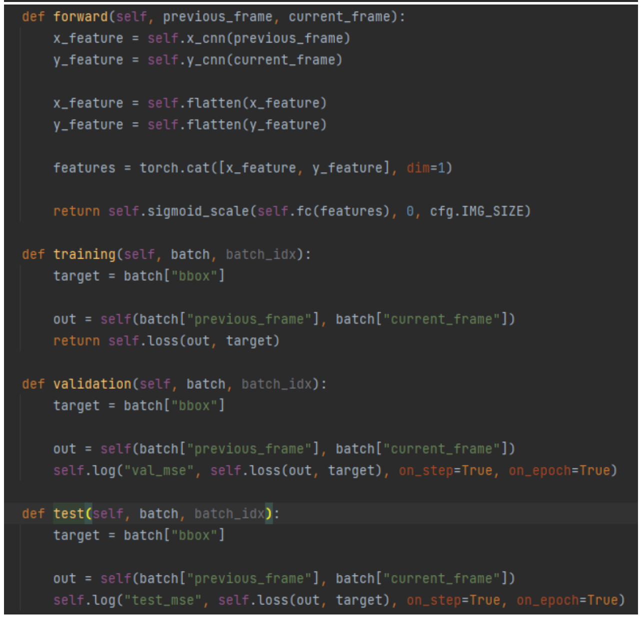


```python
class SOTModel(LightningModule):
    def __init__(self, lr=cfg.LEARNING_RATE):
        super(SOTModel, self).__init__()

        self.x_cnn = nn.Sequential(
            *(list(models.resnet34(pretrained=True).children())[:-1])
        )
        self.y_cnn = nn.Sequential(
            *(list(models.resnet34(pretrained=True).children())[:-1])
        )

        self.flatten = nn.Flatten()

        self.fc = nn.Sequential(
            nn.Linear(512 * 2, 2048),
            nn.ReLU(inplace=True),
            nn.Linear(2048, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 4),
        )

        self.lr = lr
        self.loss = nn.MSELoss()
```

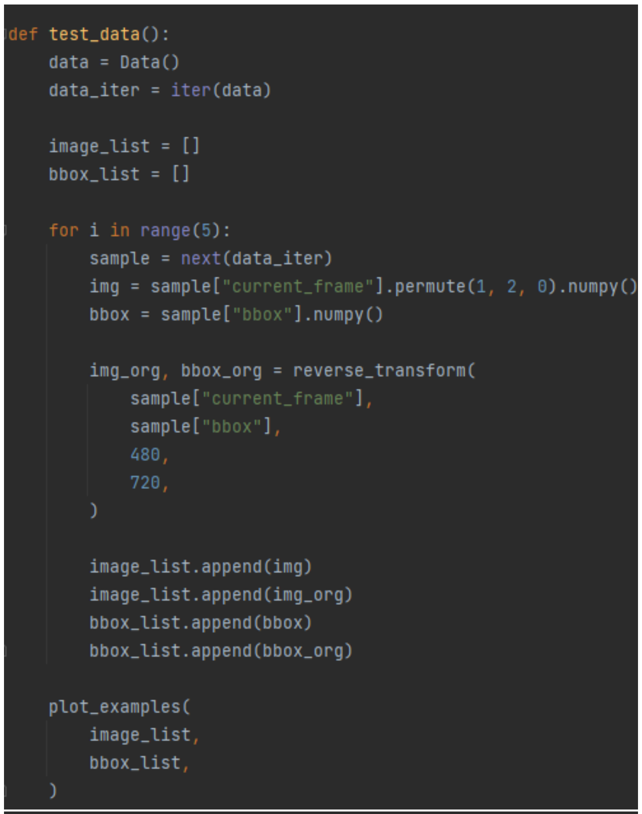
We use pytorch lightning to facilitate training and registration and therefore we write the training, validation steps and test the following. We use pytorchlightning to streamline the training and enrollment process and so we train, validate and test after writing the steps. .



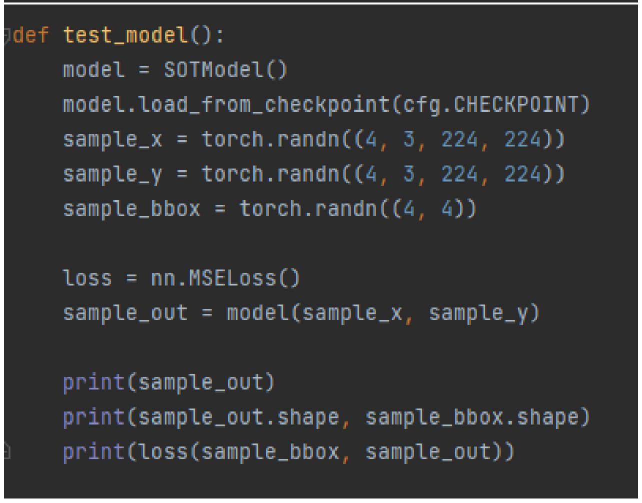
```python
def forward(self, previous_frame, current_frame):
    x_feature = self.x_cnn(previous_frame)
    y_feature = self.y_cnn(current_frame)

    x_feature = self.flatten(x_feature)
    y_feature = self.flatten(y_feature)

    features = torch.cat([x_feature, y_feature], dim=1)

    return self.sigmoid_scale(self.fc(features), 0, cfg.IMG_SIZE)
```



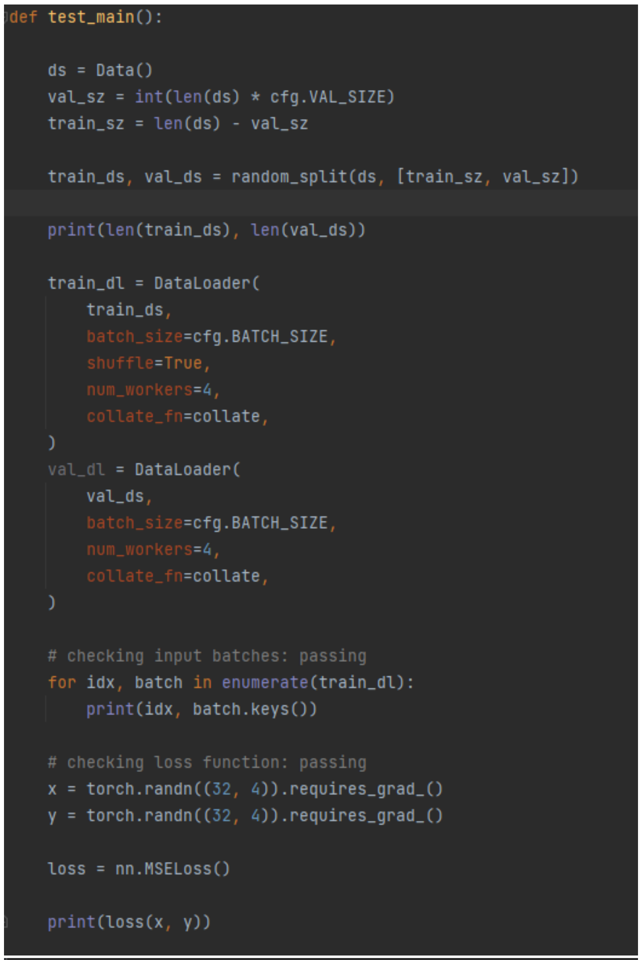
```python
def forward(self, previous_frame, current_frame):
    x_feature = self.x_cnn(previous_frame)
    y_feature = self.y_cnn(current_frame)

    x_feature = self.flatten(x_feature)
    y_feature = self.flatten(y_feature)

    features = torch.cat([x_feature, y_feature], dim=1)

    return self.sigmoid_scale(self.fc(features), 0, cfg.IMG_SIZE)

def training(self, batch, batch_idx):
    target = batch["bbox"]

    out = self(batch["previous_frame"], batch["current_frame"])
    return self.loss(out, target)

def validation(self, batch, batch_idx):
    target = batch["bbox"]

    out = self(batch["previous_frame"], batch["current_frame"])
    self.log("val_mse", self.loss(out, target), on_step=True, on_epoch=True)

def test(self, batch, batch_idx):
    target = batch["bbox"]

    out = self(batch["previous_frame"], batch["current_frame"])
    self.log("test_mse", self.loss(out, target), on_step=True, on_epoch=True)
```



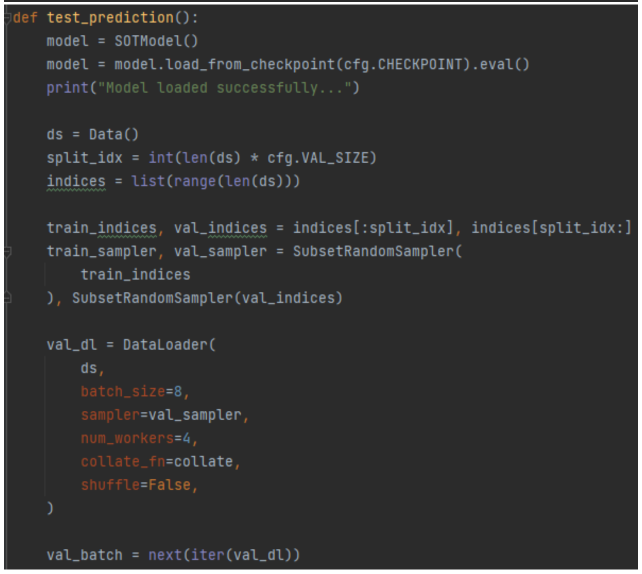
```
def test_data():
    data = Data()
    data_iter = iter(data)

    image_list = []
    bbox_list = []

    for i in range(5):
        sample = next(data_iter)
        img = sample["current_frame"].permute(1, 2, 0).numpy()
        bbox = sample["bbox"].numpy()

        img_org, bbox_org = reverse_transform(
            sample["current_frame"],
            sample["bbox"],
            480,
            720,
        )

        image_list.append(img)
        image_list.append(img_org)
        bbox_list.append(bbox)
        bbox_list.append(bbox_org)

    plot_examples(
        image_list,
        bbox_list,
    )
```



```
def test_model():
    model = SOTModel()
    model.load_from_checkpoint(cfg.CHECKPOINT)
    sample_x = torch.randn((4, 3, 224, 224))
    sample_y = torch.randn((4, 3, 224, 224))
    sample_bbox = torch.randn((4, 4))

    loss = nn.MSELoss()
    sample_out = model(sample_x, sample_y)

    print(sample_out)
    print(sample_out.shape, sample_bbox.shape)
    print(loss(sample_bbox, sample_out))
```



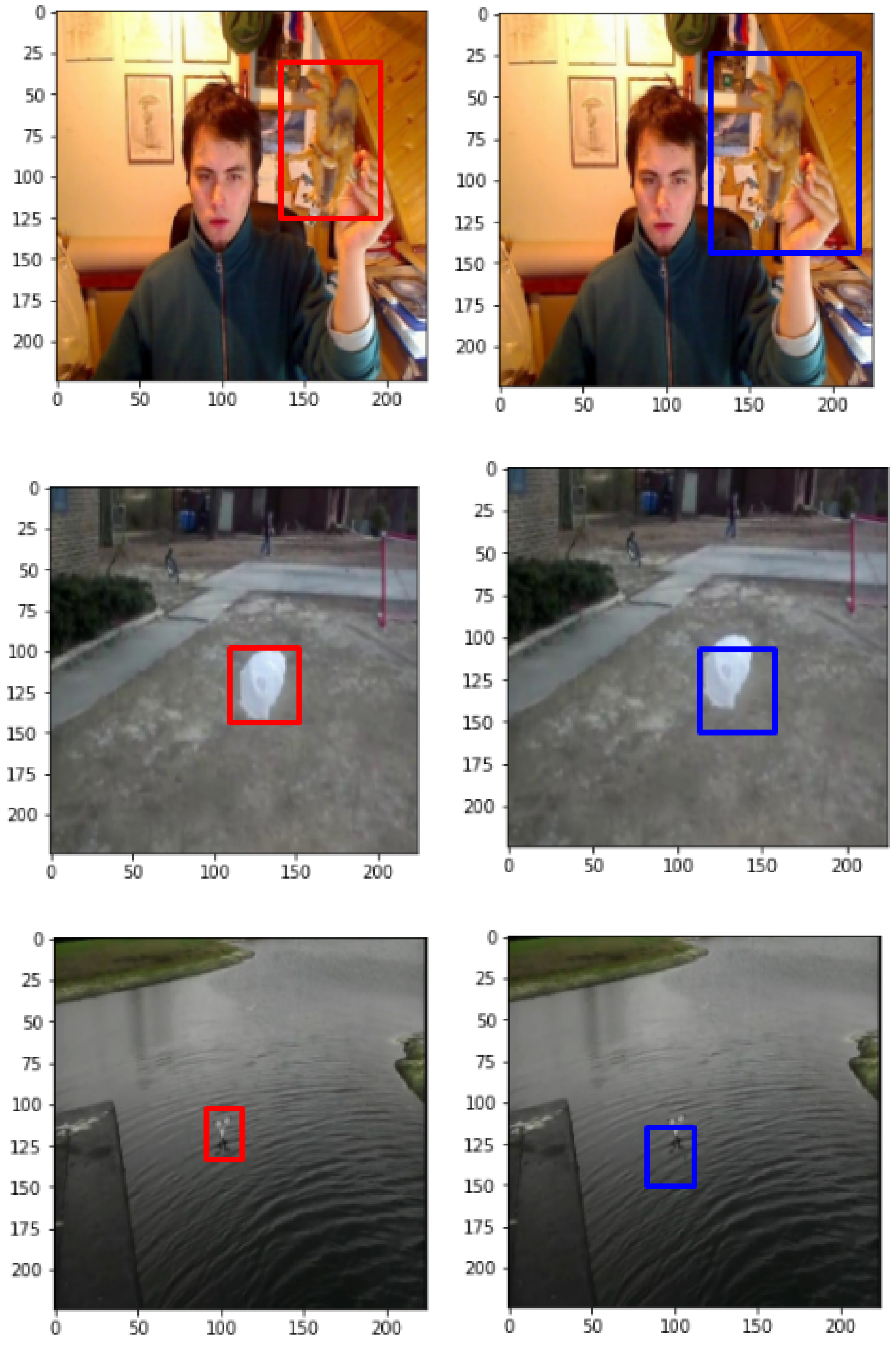
```python
def test_main():

    ds = Data()
    val_sz = int(len(ds) * cfg.VAL_SIZE)
    train_sz = len(ds) - val_sz

    train_ds, val_ds = random_split(ds, [train_sz, val_sz])

    print(len(train_ds), len(val_ds))

    train_dl = DataLoader(
        train_ds,
        batch_size=cfg.BATCH_SIZE,
        shuffle=True,
        num_workers=4,
        collate_fn=collate,
    )
    val_dl = DataLoader(
        val_ds,
        batch_size=cfg.BATCH_SIZE,
        num_workers=4,
        collate_fn=collate,
    )

    # checking input batches: passing
    for idx, batch in enumerate(train_dl):
        print(idx, batch.keys())

    # checking loss function: passing
    x = torch.randn((32, 4)).requires_grad_()
    y = torch.randn((32, 4)).requires_grad_()

    loss = nn.MSELoss()

    print(loss(x, y))
```

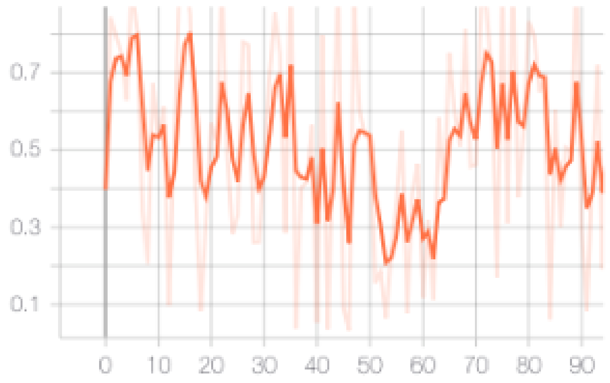


```python
def test_prediction():
    model = SOTModel()
    model = model.load_from_checkpoint(cfg.CHECKPOINT).eval()
    print("Model loaded successfully...")

    ds = Data()
    split_idx = int(len(ds) * cfg.VAL_SIZE)
    indices = list(range(len(ds)))

    train_indices, val_indices = indices[:split_idx], indices[split_idx:]
    train_sampler, val_sampler = SubsetRandomSampler(
        train_indices
    ), SubsetRandomSampler(val_indices)

    val_dl = DataLoader(
        ds,
        batch_size=8,
        sampler=val_sampler,
        num_workers=4,
        collate_fn=collate,
        shuffle=False,
    )

    val_batch = next(iter(val_dl))
```

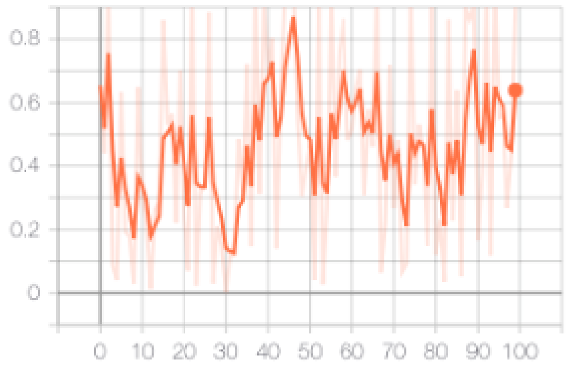
2.3 Experimental Results

Original Bounding Boxes (red) / Predicted Bounding Boxes (blue)

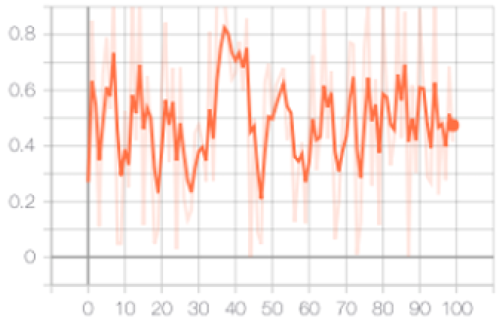




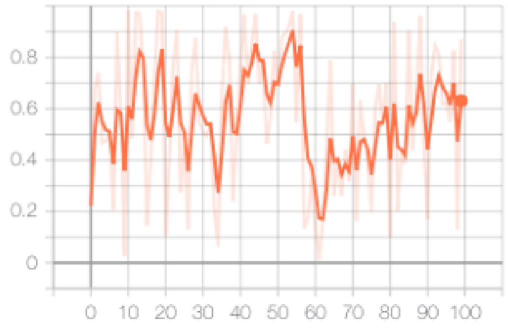
Train Loss / Epoch 50 / LR: 0.01 / BS: 64



Validation Loss / Epoch 50 / LR: 0.01 / BS: 64



Train Loss / Epoch 30 / LR: 0.001 / BS: 32



Validation Loss / Epoch 30 / LR: 0.001 / BS: 32

With a learning rate of 0.01 and a batch size of 64, the app performs best. Visually, the model is extremely impressive. Because the current model can only recognize a single object in an image with reasonable accuracy, performance can be increased further by increasing the architecture's implementation and training to detect several things in the image.

References

[1] https://medium.com/analytics-vidhya/guide-to-object-detection-using-pytorch-3925e29737b9

[2] https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html

[3]https://www.coursera.org/projects/deep-learning-with-pytorch–object-localization