Instance Segmentation using Mask RCNN

DSCI-6011-01 Deep Learning

Spring 2023

Likhita Adabala

Naveen Kumar Reddy Desireddygari

Introduction –

The goal of "instance segmentation", a unique sort of image segmentation, is to locate and identify each unique instance of something that occurs in a photograph. Instance segmentation locates all instances of a class with the additional capability to recognize separate instances of any segment class. As a result, it is sometimes referred to as functionality that combines semantic segmentation and object identification. As a result of instance segmentation, which can distinguish between several instances of the same category, distinct chairs can be distinguished by their different colors.

The availability of models in a wide range of variations and flavors enables the end user to obtain the desired result in the appropriate amount of time. Another excellent source of information with huge future potential is entertainment. There is an abundance of data accessible in this area that can make a big contribution to the Deep Learning community. Numerous new methods and strategies can be developed by fusing the realm of entertainment with state-of-the-art architecture in machine learning.

In this project, we used the Mask RCNN architecture to accomplish instance segmentation. We have created segmentation masks for each class using this architecture utilizing a number of convolutional layers. The pre-trained masks produced by the RCNN model that was trained using the supplied custom dataset are initially loaded by this model.

Data collection –

In the university's one-stop parking, surrounding Bergami Hall, and in the Maxcy Hall Parking area, we have gathered pictures of cars and sign boards. Our major goal is to use instance segmentation to find these 2 classes. A computer vision approach called instance segmentation recognizes separate items inside an image and gives each one a distinct label as well as a pixel-level mask or polygon that precisely delineates its borders.

I chose to use the MakeSense.ai software, which lets you quickly build bounding box annotations, to annotate the custom dataset. To annotate the photos more properly, you are thinking of switching to the Python LabelMe module, which provides instance segmentation capabilities. I annotated all the photographs for use in the instance segmentation model after gathering them all in one folder.

Data Loading –

The data transformations that will be applied to the photographs in the dataset are first set up by the code. It specifically fosters transformation. Make a PyTorch tensor object to convert each image. After that, a CSV file containing the image metadata and a Dataset object corresponding to the dataset's root directory

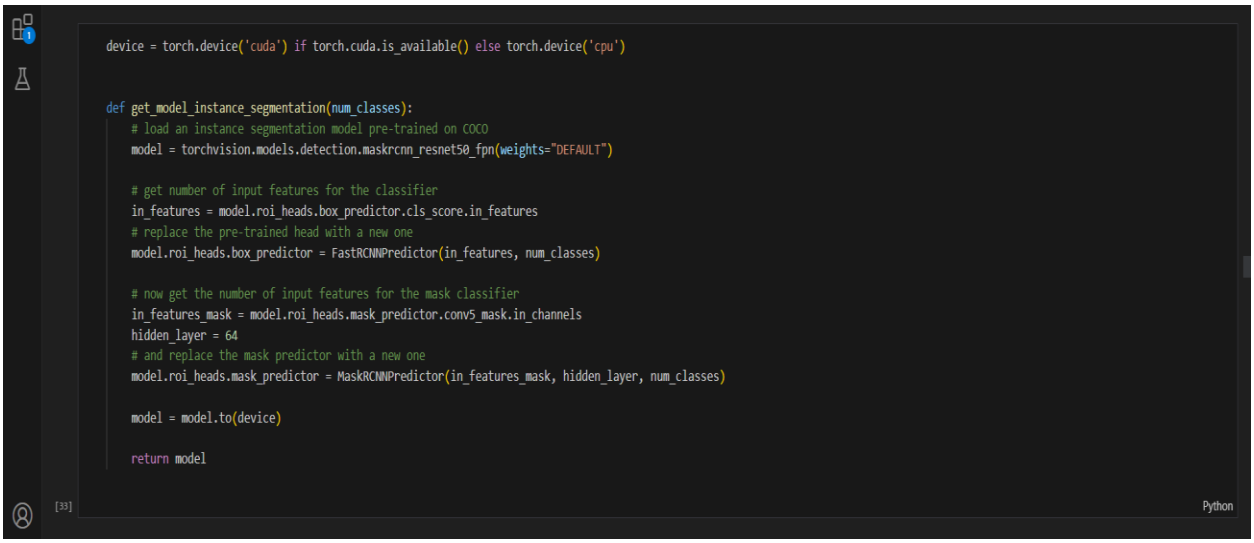
is generated. It also specifies the transformations that have previously been established. Create a representation of the adjustments to each image using an object. Every image in the collection has an index list created by the algorithm. Then, it chooses the index ranges for the training, validation, and testing splits based on the ratios specified by TRAIN_SPLIT and VAL_SPLIT and the size of the dataset.

Using these index ranges, the function creates SubsetRandomSampler objects for each split of the dataset. During training, validation, and testing, these samplers generate batches of images. During training, validation, and testing, the code creates three Data Loader objects, one for each division of the dataset, to load and preprocess photographs in batches. Using SubsetRandomSampler objects, it generates batches of images with a batch size of BATCH_SIZE.

MODEL TRAINING (Transfer Learning) :

The get_model_instance_segmentation function, which uses the Mask R-CNN architecture to construct an instance segmentation model, is defined in the code. A computer vision job called instance segmentation entails locating and categorizing each instance of an object present in a picture. The num_classes argument, which the function requires, specifies how many different object classes the model will be taught to recognize. For each object that is identified, the model will provide a collection of bounding boxes and masks. A pre-trained Mask R-CNN model that has been trained on the COCO (Common Objects in Context) dataset is then loaded by the function. This model may be used as a starting point for training a new instance segmentation model because it has already trained to recognize and categorize a large number of objects types.

CODE:



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


def get_model_instance_segmentation(num_classes):
    # load an instance segmentation model pre-trained on COCO
    model = torchvision.models.detection.maskrcnn_resnet50_fpn(weights="DEFAULT")

    # get number of input features for the classifier
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    # replace the pre-trained head with a new one
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    # now get the number of input features for the mask classifier
    in_features_mask = model.roi_heads.mask_predictor.conv5_mask.in_channels
    hidden_layer = 64
    # and replace the mask predictor with a new one
    model.roi_heads.mask_predictor = MaskRCNNPredictor(in_features_mask, hidden_layer, num_classes)

    model = model.to(device)

    return model
```

[33] Python

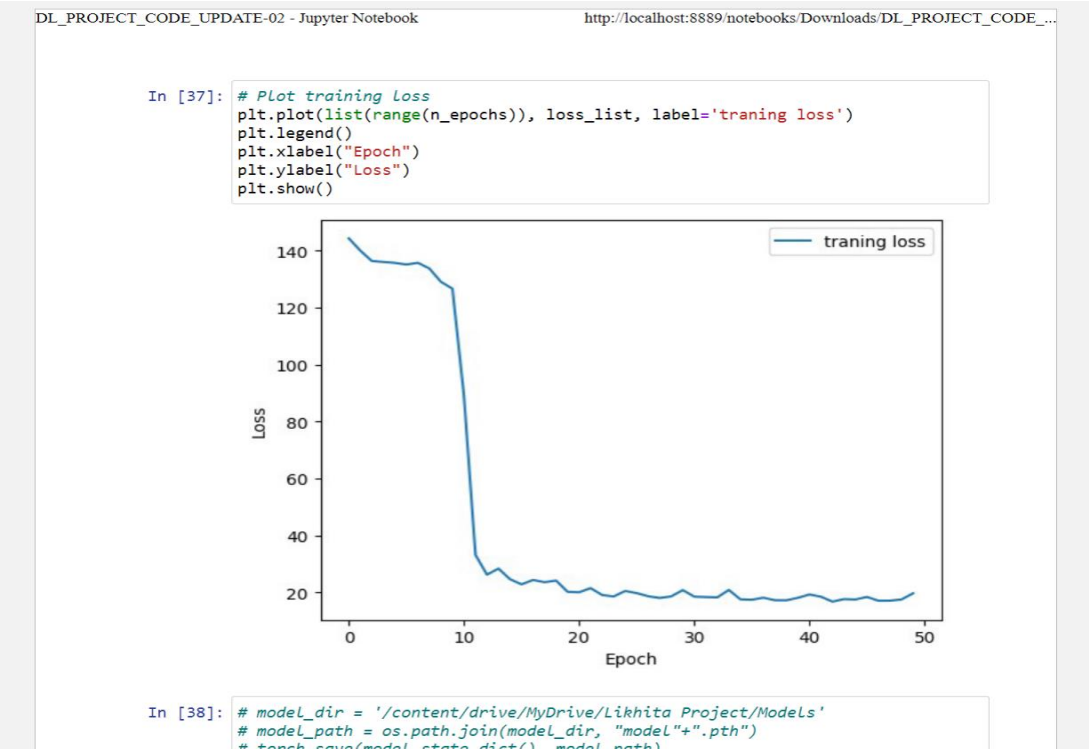
EXPERIMENTS AND RESULTS OF UPDATE-02 :

The learning rate (lr), which controls how much the optimizer modifies the parameters in

response to calculated gradients, is set to 0.00001. The optimizer moves in the same direction as

earlier stages and speeds up convergence when the momentum parameter (momentum) is set to

0.9. By penalizing big weights, the weight decay (weight decay) parameter, a type of L2 regularization that aids in preventing overfitting, is set to 0.0005. Overall, this code configures the hyperparameters for the optimizer and the instance segmentation model for training.



MODEL TRAINING (MINI NETWORK):

Transfer learning process:

The model uses transfer learning by initializing the ResNet backbone with weights pre-trained on the ImageNet dataset. This allows the model to leverage the features learned by the pre-trained network and fine-tune them for the specific task of object detection.

Configuration of each layer:

ResNet Backbone: The ResNet backbone network consists of several residual blocks, each containing multiple convolutional layers with batch normalization and

ReLU activation. The specific configuration of the ResNet architecture used in this model is ResNet-18.

Feature Pyramid Network (FPN): The FPN takes features from the ResNet backbone and generates a set of feature maps at different scales. It does this by applying a series of convolutional layers to the ResNet features and upsampling them to create a feature pyramid.

Region Proposal Network (RPN): The RPN takes the feature maps generated by the FPN and generates region proposals, which are areas of the image that are likely to contain objects. It does this by applying a set of convolutional layers and anchor boxes to the feature maps.

ROI Align Pooling: The ROI Align Pooling layer takes the region proposals generated by the RPN and extracts features from the feature maps at the corresponding locations. It does this by aligning the feature maps with the region proposals using bilinear interpolation.

Box and Mask Heads: The box head takes the ROI-pooled features and passes them through a series of fully connected layers to predict the bounding box coordinates and class probabilities. The mask head takes the ROI-pooled features and passes them through a series of convolutional layers to predict a binary mask for each class.

Overall, this model uses a combination of convolutional layers, pooling layers, and fully connected layers to perform object detection on input images. It leverages transfer learning by using a pre-trained ResNet backbone and fine-tuning it for the specific task for segmentation task.

CODE :

```
import torchvision.models as models
]
class MiniNet(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.num_classes = num_classes

        # backbone network
        self.backbone = models.resnet18(pretrained=True)

        # Feature Pyramid Network (FPN)
        self.fpn = models.detection.FPN([64, 128, 256, 512], 256)

        # region proposal network (RPN)
        self.rpn = models.detection.rpn.AnchorGenerator(
            sizes=((32, 64, 128, 256, 512),),
            aspect_ratios=((0.5, 1.0, 2.0),)
        )

        # ROI align pooling
        self.roi_pool = models.detection.roi_heads.RoIAlign(output_size=(7, 7), sampling_ratio=2)

        # box and mask heads
        self.box_head = nn.Sequential(
            nn.Linear(256 * 7 * 7, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 1024),
            nn.ReLU(inplace=True)
        )

        self.box_predictor = nn.Sequential(
            nn.Linear(1024, num_classes * 4)
        )

        self.mask_head = nn.Sequential(
```

```
        self.mask_head = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True)
        )

        self.mask_predictor = nn.Sequential(
            nn.ConvTranspose2d(256, 256, kernel_size=2, stride=2),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(256, num_classes, kernel_size=2, stride=2),
            nn.ReLU(inplace=True)
        )
        # initialize conv3 layers in the backbone with kaiming_normal initialization
        for layer in self.backbone.layer3:
            if isinstance(layer, nn.Conv2d):
                init.kaiming_normal_(layer.weight, mode='fan_out', nonlinearity='relu')
                if layer.bias is not None:
                    nn.init.constant_(layer.bias, 0)
    def forward(self, images, targets):
        # backbone features
        features = self.backbone(images)

        # FPN features
        features = self.fpn(features)

        # RPN proposals
        proposals = self.rpn(features)

        # ROI pooled features
        box_features = self.roi_pool(features, proposals)
```

```
            nn.ConvTranspose2d(256, num_classes, kernel_size=2, stride=2),
            nn.ReLU(inplace=True)
        )
        # initialize conv3 layers in the backbone with kaiming_normal initialization
        for layer in self.backbone.layer3:
            if isinstance(layer, nn.Conv2d):
                init.kaiming_normal_(layer.weight, mode='fan_out', nonlinearity='relu')
                if layer.bias is not None:
                    nn.init.constant_(layer.bias, 0)
    def forward(self, images, targets):
        # backbone features
        features = self.backbone(images)

        # FPN features
        features = self.fpn(features)

        # RPN proposals
        proposals = self.rpn(features)

        # ROI pooled features
        box_features = self.roi_pool(features, proposals)
        mask_features = self.roi_pool(features, proposals)

        # box and mask heads
        box_features = self.box_head(box_features.flatten(start_dim=1))
        class_logits = self.box_predictor(box_features).view(-1, self.num_classes, 4)

        mask_features = self.mask_head(mask_features)
        mask_logits = self.mask_predictor(mask_features)

        # output dictionary
        output = {
            "loss_box_reg": class_logits,
            "loss_mask": mask_logits
        }
```

Experiments and Results:

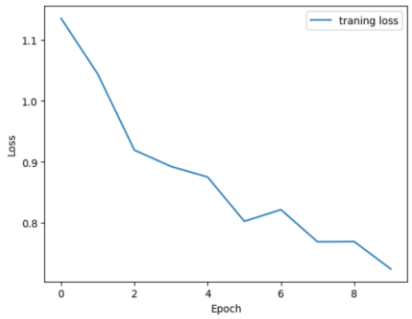
HYPER PARAMETER VALUES:

We have taken almost the similar values which we used for Transfer learning within the 10 epochs.

```
[ ] params = [p for p in model.parameters() if p.requires_grad]
    optimizer = torch.optim.SGD(params, lr=0.00001, momentum=0.9, weight_decay=0.0005)
```

EPOCHS :10

```
# Plot training loss
plt.plot(list(range(n_epochs)), loss_list, label='traning loss')
plt.legend()
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```



RESULTED OUTPUT:

Instead of an image, I have printed a tensor.

Conclusion-

Finally, I want to conclude that, after constructing the mini network I have iterated 10 tensors. I have produced a blank tensor. Which I couldn't be able to construct. This isn't a pretrained model we used instead we used a backbone network Rest18 and then I constructed a few convolutional layers for the next RO1 AND RPN features using foreground and background coordinates.

DATASET LINK :

ANNOTATED DATASET LINK-

https://drive.google.com/drive/folders/1wkIgGVsD3gxXcWUk1ySGumnSxU3tSquM

ORGINIAL DATASET:

https://drive.google.com/drive/folders/1Atb4Sn7pgKx0jnRmk4FplhdCN7WmWVAq?usp=share_link