Instance Segmentation

Final Project Report

Pravallika Muraboyina & Chandrasheker Bassetti

Introduction:

Instance segmentation is a task that involves identification of individual objects within an image and the allocation of each pixel to a specific object instance. When compared to semantic segmentation, which classifies pixels into categories, instance segmentation offers more specific information about the location and boundaries of each object. This is important for a variety of applications like object recognition, medical imaging and autonomous driving.

Methodology:

Based on the user's needs, this instance segmentation model will offer a means to extract useful information from a given image. It functions as described in the following steps:

1. To begin with, we initially upload the sample images that we have collected into a cloud platform for storage.

2. All the images are next annotated with specific labels related to the user requirements.

3. These images are trained using Mask RCNN models.

3. Finally, output is retrieved as the masked images.

Dataset and Data Pre-processing

The sample images for this project were collected from different locations around University of New Haven. It includes classrooms, library, market place, office locations, gymnasium, parking, etc. These images cover a wide range of objects such as people, benches, cars, fire extinguishers, charger horse, animals, etc. All these images are annotated using Label Studio in by following the JSON format which is used in the COCO dataset. All the images are currently placed in the below cloud link.

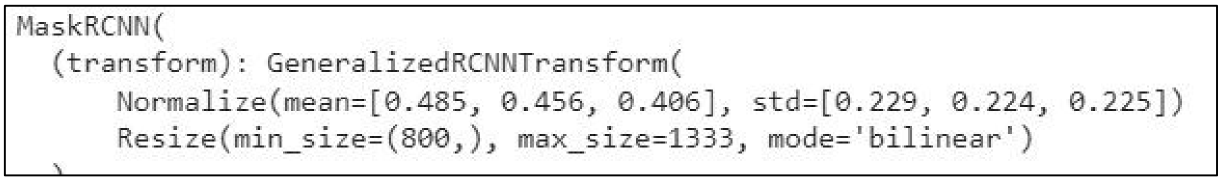
https://unhnewhaven-my.sharepoint.com/:f:/g/personal/pmura1_unh_newhaven_edu/EpUthGe3nFZOtqv3rFsvWi0BtGwz4FxtoEhQ2h56e7BKug?e=8ZZ5W3

Once the images are annotated previously in update 1, we implemented transformation functions. These functions are basically used to convert the image into tensor format and to perform resizing. It is later used to normalize functions on our input images.

As we desire the output images in the rectangular format, image resizing is performed. Here, we considered masked images in the binary format for better evaluations. We also added R,G,B masked images trying using different methods to get the best outputs.

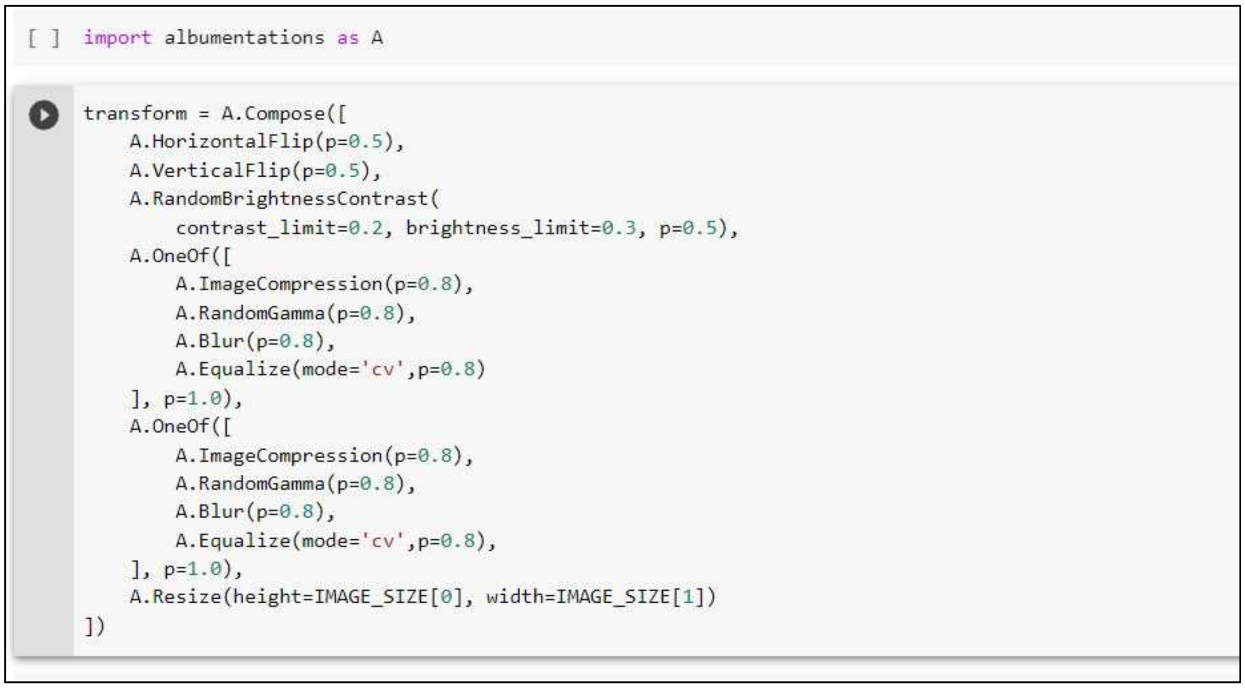
Data Pre-processing

For our project, we utilized a pre-trained model that employed Normalize and Resize Transformations. To achieve this, we incorporated the albumentation library, known for its speed and versatility in image augmentation. This powerful library facilitated various transformations, including image compression, flipping, and random image quality enhancements. It also enabled us to efficiently perform resizing and normalizing operations on our images, which had a size of 1024 x 1024 pixels. By implementing data augmentation techniques, we aimed to mitigate the risk of overfitting and minimize computational complexities during our analysis.



```
MaskRCNN(
  (transform): GeneralizedRCNNTransform(
      Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
      Resize(min_size=(800,), max_size=1333, mode='bilinear')
```

Pre-trained Model Transformation

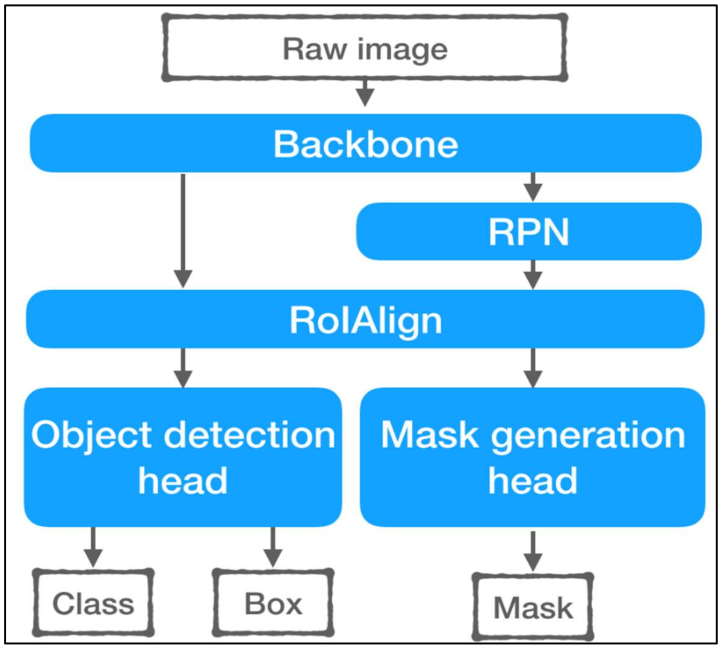


```
[ ] import albumentations as A

transform = A.Compose([
    A.HorizontalFlip(p=0.5),
    A.VerticalFlip(p=0.5),
    A.RandomBrightnessContrast(
        contrast_limit=0.2, brightness_limit=0.3, p=0.5),
    A.OneOf([
        A.ImageCompression(p=0.8),
        A.RandomGamma(p=0.8),
        A.Blur(p=0.8),
        A.Equalize(mode='cv',p=0.8)
    ], p=1.0),
    A.OneOf([
        A.ImageCompression(p=0.8),
        A.RandomGamma(p=0.8),
        A.Blur(p=0.8),
        A.Equalize(mode='cv',p=0.8),
    ], p=1.0),
    A.Resize(height=IMAGE_SIZE[0], width=IMAGE_SIZE[1])
])
```

Data Augumentations

Transfer Learning

Model Architecture:



Transfer learning is a machine learning method where a pre-trained model is utilized as a starting point for a new model trained on a separate but related task or dataset. The pre-trained model is utilized as a starting point rather than starting the training process from scratch, and the model's weights are adjusted on the fresh dataset. Transfer learning can result in quicker convergence, higher accuracy, and a reduction in training time and resources by using the information and representation acquired by a model that has already been trained.

The different modules of Masked RCNN are:

Backbone Network: The backbone network retrieves high-level characteristics from the input pictures, often using a convolutional neural network (CNN).

Region Proposal Network (RPN): Using anchor boxes, the RPN creates probable bounding box locations for items in the picture, assigning probability to each area suggestion.

Mask Prediction Network: For accurate pixel-level segmentation of objects, the mask prediction network conducts object classification and produces binary instance masks.

Multi-Task Loss Function: Bounding box regression, object classification, and mask prediction are all concurrently learned during training using an optimized multi-task loss function.

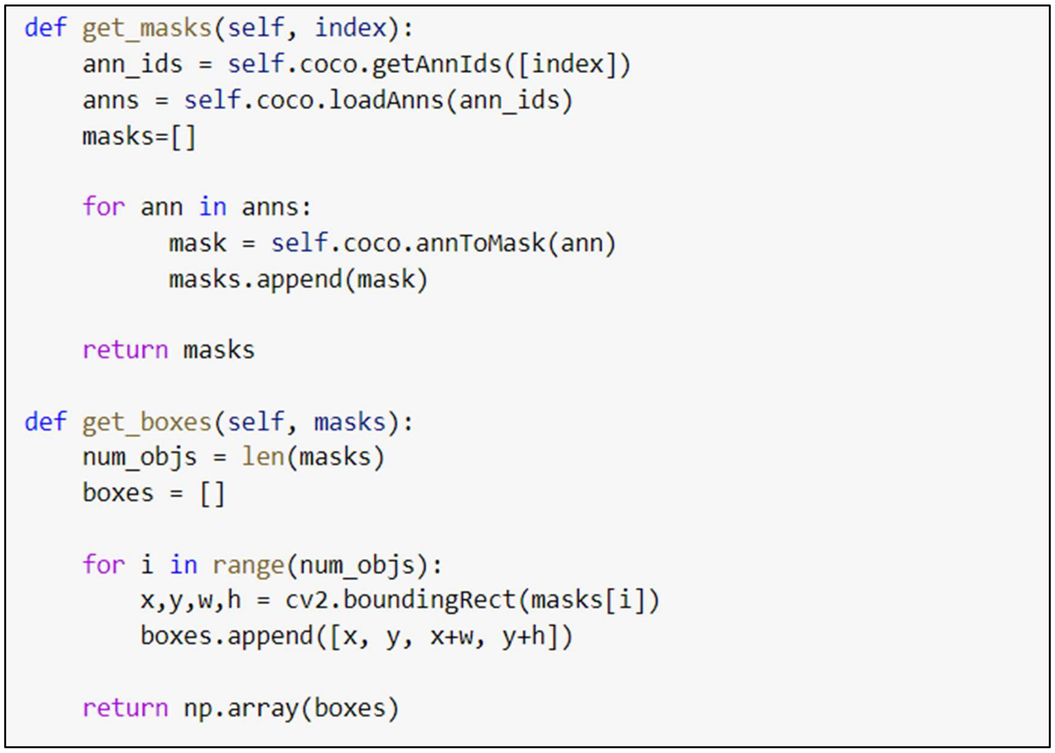
Object Detection: Mask R-CNN is frequently used to precisely recognize objects in images and determine their existence and placement.

Instance Segmentation: Mask R-CNN goes beyond object identification with its mask prediction capabilities and offers precise segmentation masks for specific items in an image.

Objective Function:

The main objective of the MaskRCNN model is to find the class label for each instance in the image, the bounding box for the detected instance, and generate the mask for the same.

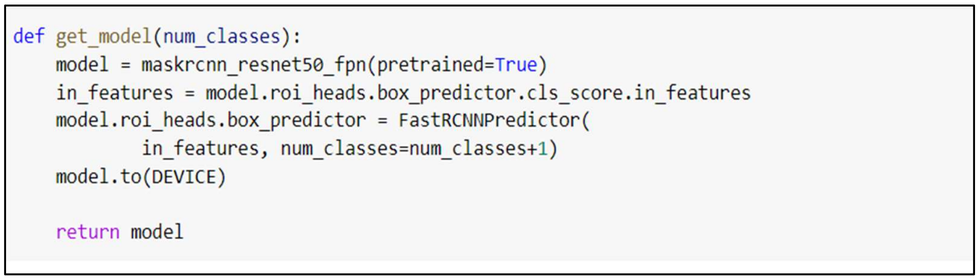
The below code reads the masks and bounding box values from the loaded json annotation files.



```
def get_masks(self, index):
    ann_ids = self.coco.getAnnIds([index])
    anns = self.coco.loadAnns(ann_ids)
    masks=[]

    for ann in anns:
          mask = self.coco.annToMask(ann)
          masks.append(mask)

    return masks

def get_boxes(self, masks):
    num_objs = len(masks)
    boxes = []

    for i in range(num_objs):
        x,y,w,h = cv2.boundingRect(masks[i])
        boxes.append([x, y, x+w, y+h])

    return np.array(boxes)
```

Parameters:

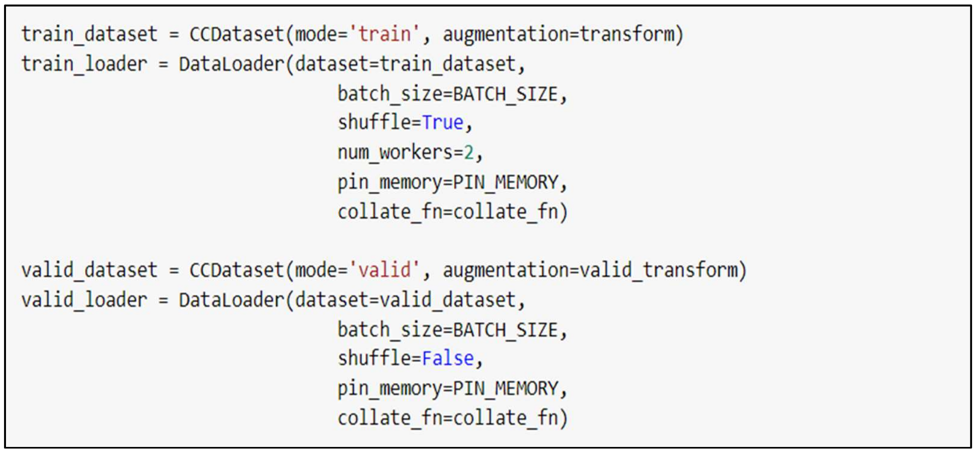
Considered the pre-trained weights of maskrcnn_resnet50_fpn for mask generation. And for bounding box detection FastRCNNPredictor model with num_classes =10. Below is the code for the same.



```
def get_model(num_classes):
    model = maskrcnn_resnet50_fpn(pretrained=True)
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    model.roi_heads.box_predictor = FastRCNNPredictor(
            in_features, num_classes=num_classes+1)
    model.to(DEVICE)

    return model
```

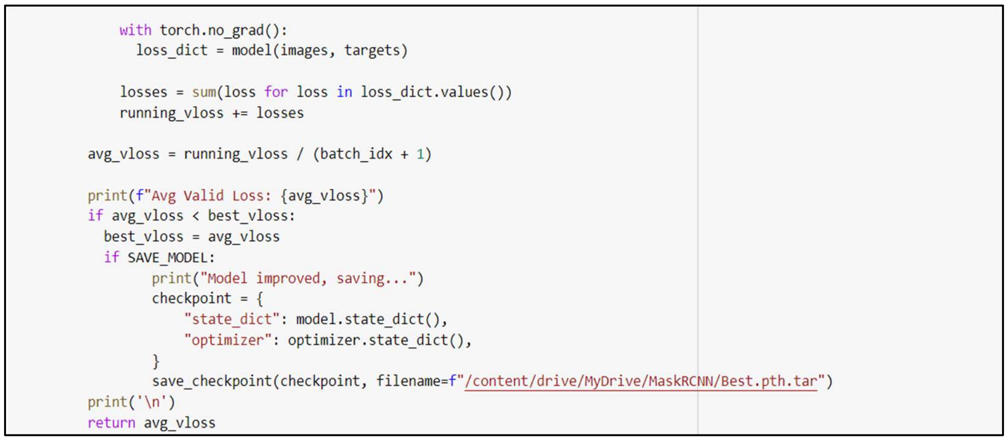
Experiments and Results:

Fine Tuning: This model is fine-tuned by limiting the number of class labels to 10. Also, class labels and bounding boxes for pre-trained models are loaded from the annotated JSON files.



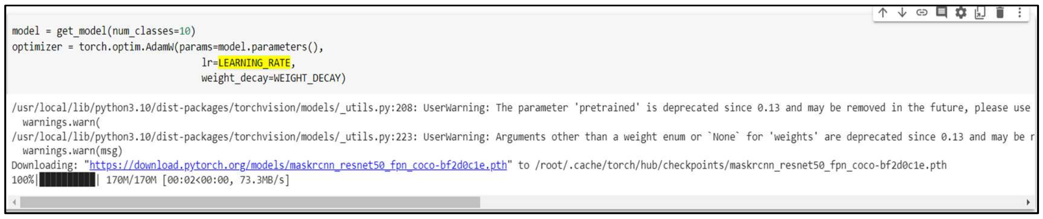
```
train_dataset = CCDataset(mode='train', augmentation=transform)
train_loader = DataLoader(dataset=train_dataset,
                          batch_size=BATCH_SIZE,
                          shuffle=True,
                          num_workers=2,
                          pin_memory=PIN_MEMORY,
                          collate_fn=collate_fn)

valid_dataset = CCDataset(mode='valid', augmentation=valid_transform)
valid_loader = DataLoader(dataset=valid_dataset,
                          batch_size=BATCH_SIZE,
                          shuffle=False,
                          pin_memory=PIN_MEMORY,
                          collate_fn=collate_fn)
```

Hyper-parameter selection: Loss at each epoch is calculated to find the model state where loss is minimum. At this point state of the model, hyper-parameters, and parameters are saved for further use of the model.



```
        with torch.no_grad():
          loss_dict = model(images, targets)

        losses = sum(loss for loss in loss_dict.values())
        running_vloss += losses

    avg_vloss = running_vloss / (batch_idx + 1)

    print(f"Avg Valid Loss: {avg_vloss}")
    if avg_vloss < best_vloss:
      best_vloss = avg_vloss
      if SAVE_MODEL:
            print("Model improved, saving...")
            checkpoint = {
                "state_dict": model.state_dict(),
                "optimizer": optimizer.state_dict(),
            }
            save_checkpoint(checkpoint, filename=f"/content/drive/MyDrive/MaskRCNN/Best.pth.tar")
    print('\n')
    return avg_vloss
```

Learning Rate Decay policy: We manually tested for the best learning rate and gave that as a fixed input to the optimizer function.



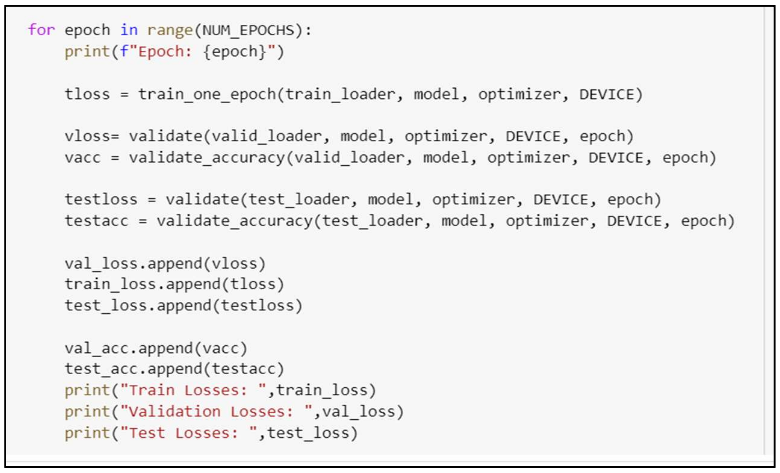
```
model = get_model(num_classes=10)
optimizer = torch.optim.AdamW(params=model.parameters(),
                              lr=LEARNING_RATE,
                              weight_decay=WEIGHT_DECAY)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be r
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/maskrcnn_resnet50_fpn_coco-bf2d0c1e.pth" to /root/.cache/torch/hub/checkpoints/maskrcnn_resnet50_fpn_coco-bf2d0c1e.pth
100%|██████████| 170M/170M [00:02<00:00, 73.3MB/s]
```

Model Selection: Model selection is also done based on loss at each epoch. As shown in the hyper-parameter tuning above, at the same stage the best model is also saved.

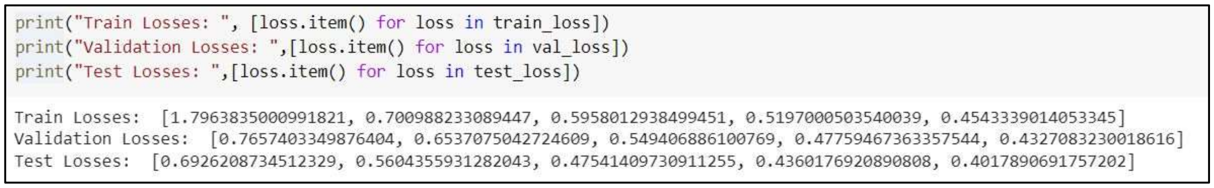
'state_dict: model.state.dict()'

Evaluation: Losses for train dataset, validation dataset and test dataset are calculated to observe the model performance

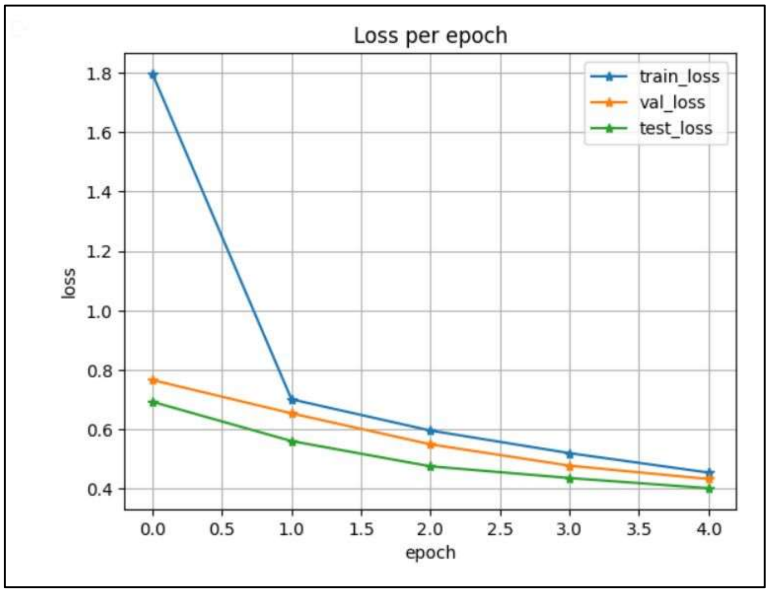


```
for epoch in range(NUM_EPOCHS):
    print(f"Epoch: {epoch}")

    tloss = train_one_epoch(train_loader, model, optimizer, DEVICE)

    vloss= validate(valid_loader, model, optimizer, DEVICE, epoch)
    vacc = validate_accuracy(valid_loader, model, optimizer, DEVICE, epoch)

    testloss = validate(test_loader, model, optimizer, DEVICE, epoch)
    testacc = validate_accuracy(test_loader, model, optimizer, DEVICE, epoch)

    val_loss.append(vloss)
    train_loss.append(tloss)
    test_loss.append(testloss)

    val_acc.append(vacc)
    test_acc.append(testacc)
    print("Train Losses: ",train_loss)
    print("Validation Losses: ",val_loss)
    print("Test Losses: ",test_loss)
```

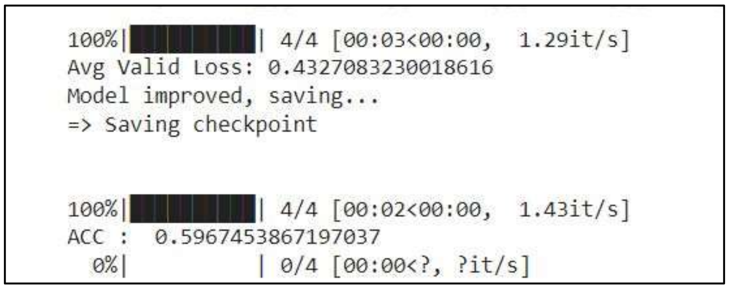
Losses: Loss values are given as



```
print("Train Losses: ", [loss.item() for loss in train_loss])
print("Validation Losses: ",[loss.item() for loss in val_loss])
print("Test Losses: ",[loss.item() for loss in test_loss])

Train Losses:  [1.7963835000991821, 0.700988233089447, 0.5958012938499451, 0.5197000503540039, 0.4543339014053345]
Validation Losses:  [0.7657403349876404, 0.6537075042724609, 0.549406886100769, 0.47759467363357544, 0.4327083230018616]
Test Losses:  [0.6926208734512329, 0.5604355931282043, 0.47541409730911255, 0.4360176920890808, 0.4017890691757202]
```



Accuracy: Accuracy is given as



```
100%|██████████| 4/4 [00:03<00:00,  1.29it/s]
Avg Valid Loss: 0.4327083230018616
Model improved, saving...
=> Saving checkpoint


100%|██████████| 4/4 [00:02<00:00,  1.43it/s]
ACC :  0.5967453867197037
  0%|          | 0/4 [00:00<?, ?it/s]
```

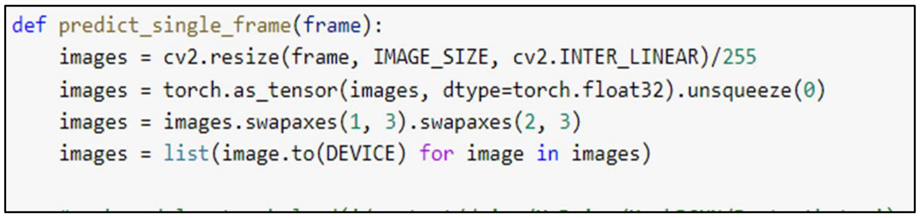
Deployment:

Library: Gradio

Gradio is a Python toolkit that makes it easier to build and use web-based user interfaces for machine learning models. It accepts a variety of input formats, including text, photos, audio, and video, and has an easy user interface. Gradio manages the interface's interaction with models, making it simple for users to interact with the model. With just one line of code, it streamlines deployment and offers UI components for customization. By bridging the gap between ML models and end users, Gradio increases the accessibility of model exploration and deployment.

As our model in instance segmentation both the input from the gradio and output to the gradio will be an image.

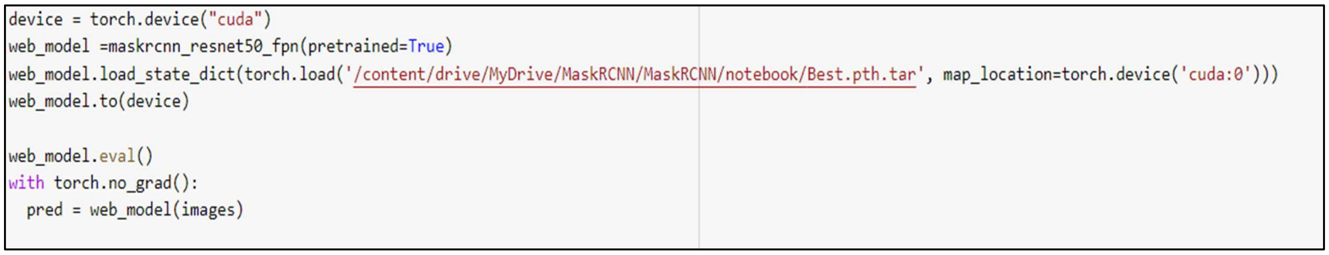
Before sending this gradio image to the model we are using some transformations as part of *Data Pre-processing.*



```python
def predict_single_frame(frame):
    images = cv2.resize(frame, IMAGE_SIZE, cv2.INTER_LINEAR)/255
    images = torch.as_tensor(images, dtype=torch.float32).unsqueeze(0)
    images = images.swapaxes(1, 3).swapaxes(2, 3)
    images = list(image.to(DEVICE) for image in images)
```

Transformation functions

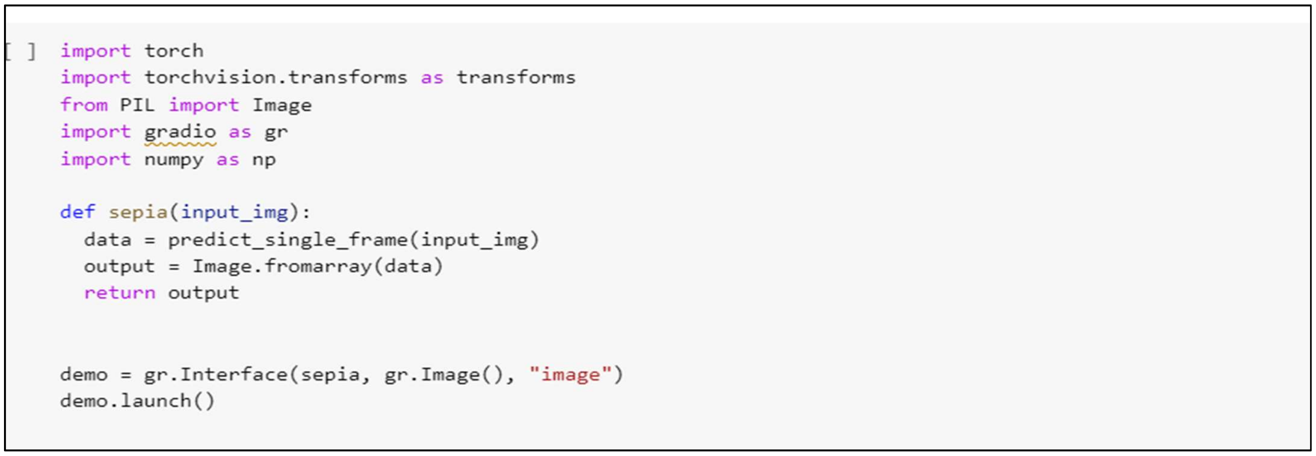
Once the image is transformed it is given as input to the model.



```python
device = torch.device("cuda")
web_model =maskrcnn_resnet50_fpn(pretrained=True)
web_model.load_state_dict(torch.load('/content/drive/MyDrive/MaskRCNN/MaskRCNN/notebook/Best.pth.tar', map_location=torch.device('cuda:0')))
web_model.to(device)

web_model.eval()
with torch.no_grad():
  pred = web_model(images)
```

Model initiation with input

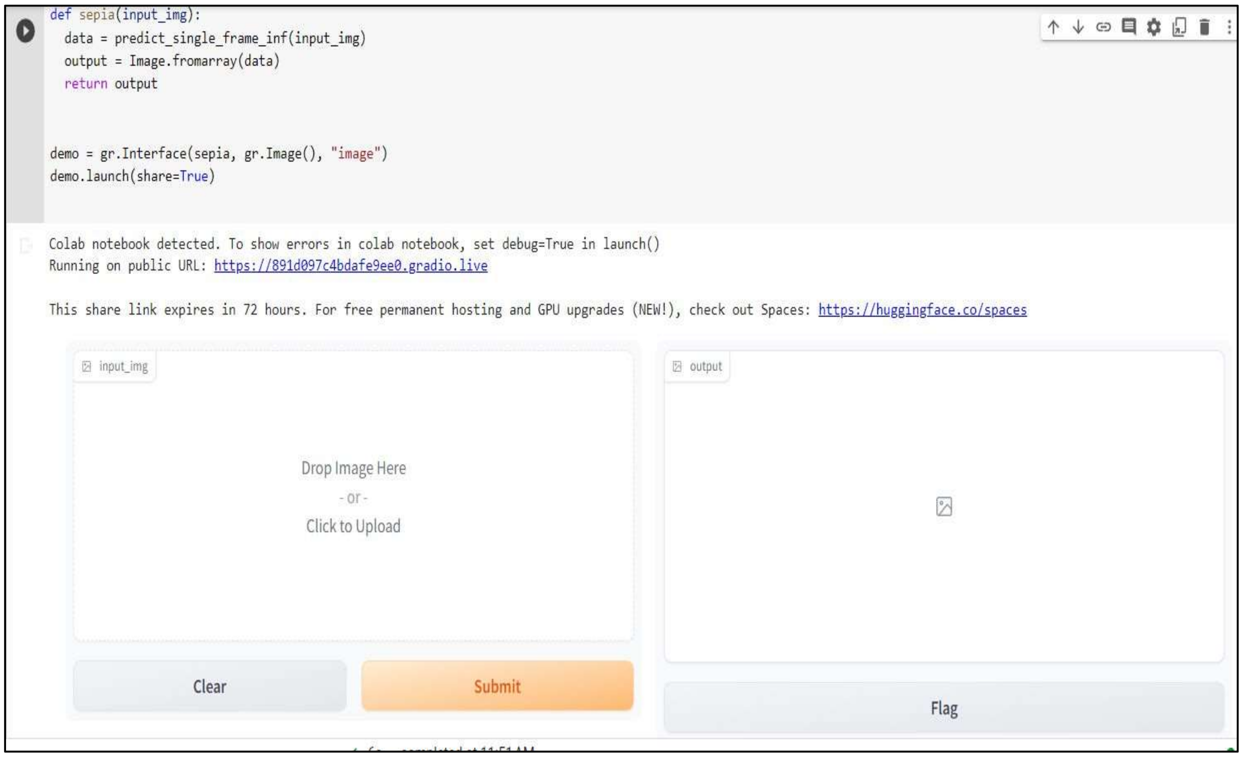
Transforming and sending outputs to gradio



```python
[ ]  import torch
     import torchvision.transforms as transforms
     from PIL import Image
     import gradio as gr
     import numpy as np

     def sepia(input_img):
       data = predict_single_frame(input_img)
       output = Image.fromarray(data)
       return output


     demo = gr.Interface(sepia, gr.Image(), "image")
     demo.launch()
```

Output Transformations

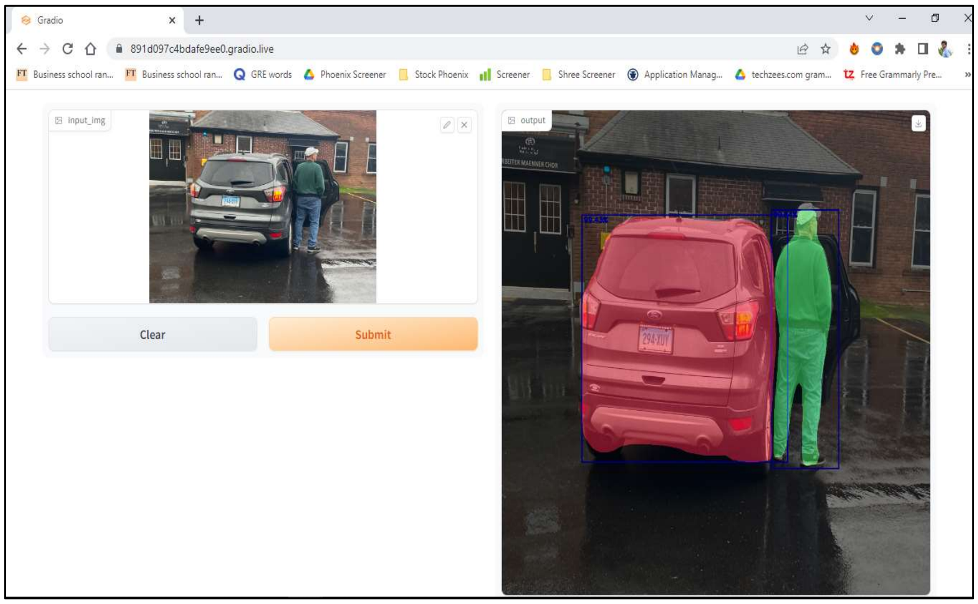
A link is generated to test the model.



```
def sepia(input_img):
  data = predict_single_frame_inf(input_img)
  output = Image.fromarray(data)
  return output


demo = gr.Interface(sepia, gr.Image(), "image")
demo.launch(share=True)
```

```
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
Running on public URL: https://891d097c4bdafe9ee0.gradio.live

This share link expires in 72 hours. For free permanent hosting and GPU upgrades (NEW!), check out Spaces: https://huggingface.co/spaces
```

Deployment Results:

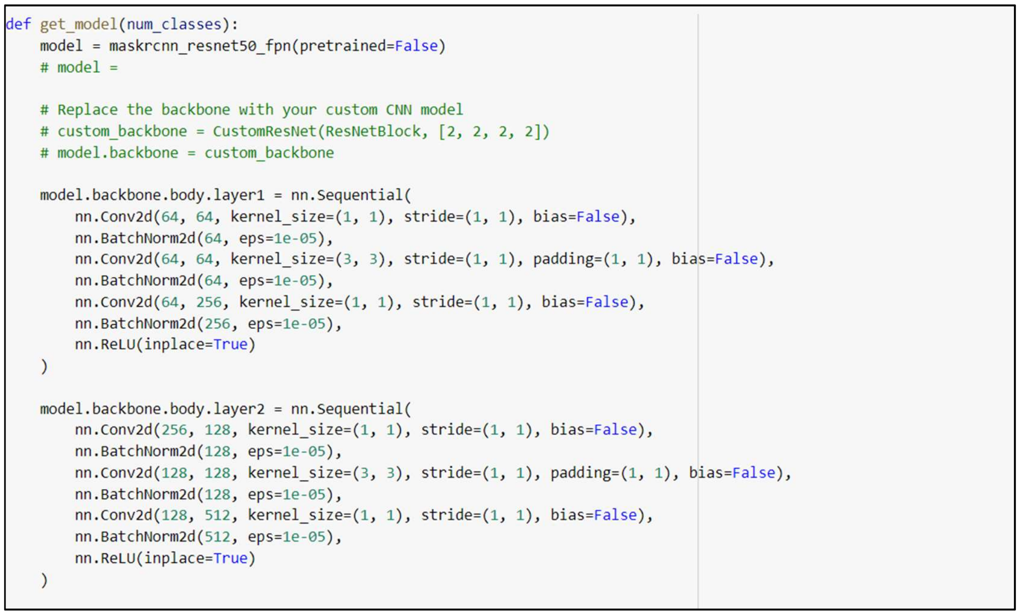


Mini Network

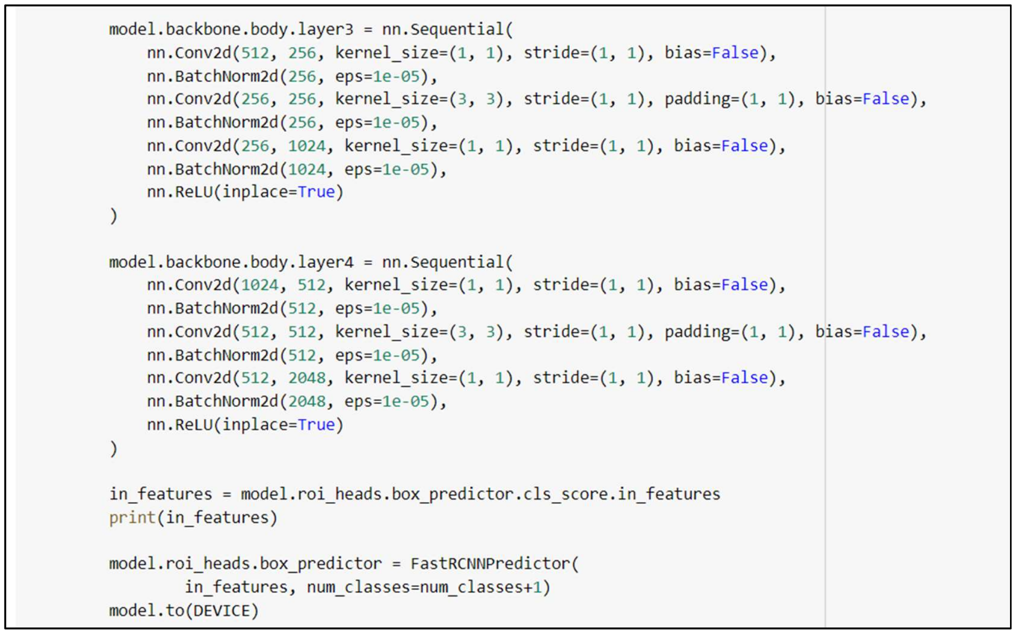
Backbone architecture:

MaskRCNN model is initiated with pre-trained weight= False. A mini network is created for the backbone layer structure, and this was used in the model architecture instead of the pre-trained model backbone.

We have created a total of 4 sequential layers as shown below.



```
def get_model(num_classes):
    model = maskrcnn_resnet50_fpn(pretrained=False)
    # model =

    # Replace the backbone with your custom CNN model
    # custom_backbone = CustomResNet(ResNetBlock, [2, 2, 2, 2])
    # model.backbone = custom_backbone

    model.backbone.body.layer1 = nn.Sequential(
        nn.Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False),
        nn.BatchNorm2d(64, eps=1e-05),
        nn.Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False),
        nn.BatchNorm2d(64, eps=1e-05),
        nn.Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False),
        nn.BatchNorm2d(256, eps=1e-05),
        nn.ReLU(inplace=True)
    )

    model.backbone.body.layer2 = nn.Sequential(
        nn.Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False),
        nn.BatchNorm2d(128, eps=1e-05),
        nn.Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False),
        nn.BatchNorm2d(128, eps=1e-05),
        nn.Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False),
        nn.BatchNorm2d(512, eps=1e-05),
        nn.ReLU(inplace=True)
    )
```

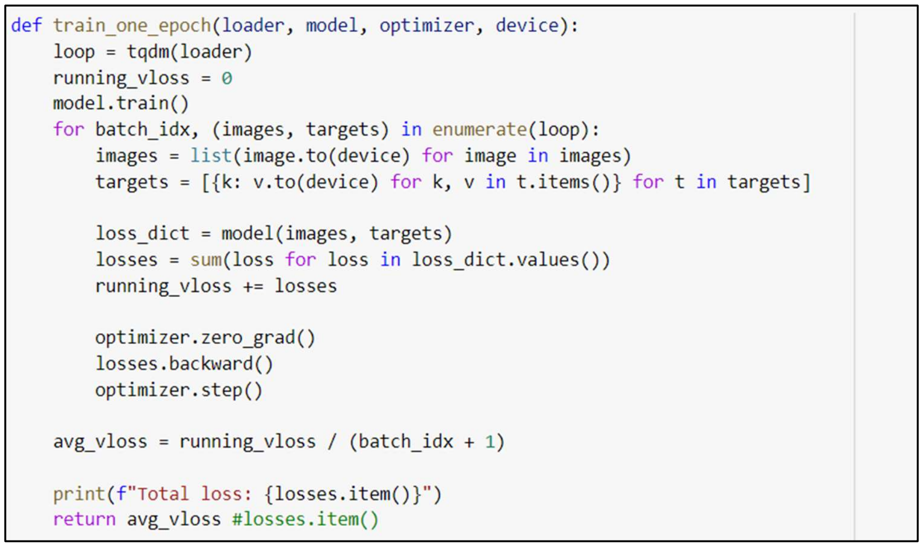


```
    model.backbone.body.layer3 = nn.Sequential(
        nn.Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False),
        nn.BatchNorm2d(256, eps=1e-05),
        nn.Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False),
        nn.BatchNorm2d(256, eps=1e-05),
        nn.Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False),
        nn.BatchNorm2d(1024, eps=1e-05),
        nn.ReLU(inplace=True)
    )

    model.backbone.body.layer4 = nn.Sequential(
        nn.Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False),
        nn.BatchNorm2d(512, eps=1e-05),
        nn.Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False),
        nn.BatchNorm2d(512, eps=1e-05),
        nn.Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False),
        nn.BatchNorm2d(2048, eps=1e-05),
        nn.ReLU(inplace=True)
    )

    in_features = model.roi_heads.box_predictor.cls_score.in_features
    print(in_features)

    model.roi_heads.box_predictor = FastRCNNPredictor(
            in_features, num_classes=num_classes+1)
    model.to(DEVICE)
```

Backbone architecture

Objective Function:

The main objective of the model is to generate a bounding box for each instance in the image and class and mask for the same instance.

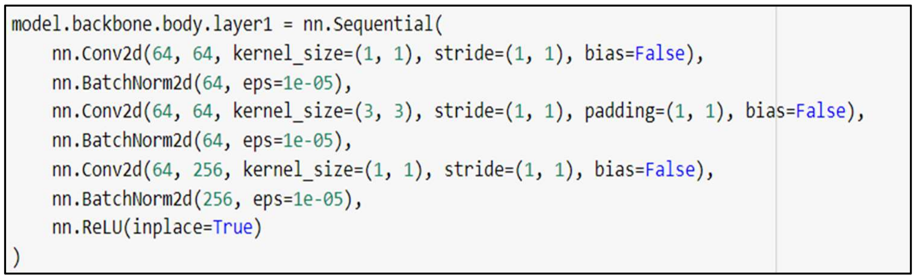


```
def train_one_epoch(loader, model, optimizer, device):
    loop = tqdm(loader)
    running_vloss = 0
    model.train()
    for batch_idx, (images, targets) in enumerate(loop):
        images = list(image.to(device) for image in images)
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        loss_dict = model(images, targets)
        losses = sum(loss for loss in loss_dict.values())
        running_vloss += losses

        optimizer.zero_grad()
        losses.backward()
        optimizer.step()

    avg_vloss = running_vloss / (batch_idx + 1)

    print(f"Total loss: {losses.item()}")
    return avg_vloss #losses.item()
```

Parameters of Inputs and Outputs:

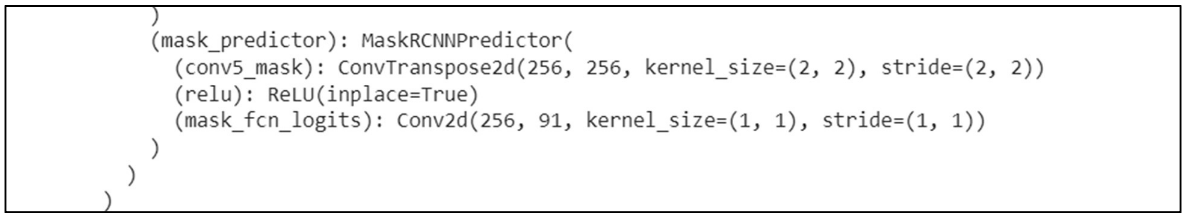
Input Layer Parameters:

Image of size: 64 X 64
Kernel_size: (1,1)
Stride: (1,1)
Activation function: ReLU



```
model.backbone.body.layer1 = nn.Sequential(
    nn.Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False),
    nn.BatchNorm2d(64, eps=1e-05),
    nn.Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False),
    nn.BatchNorm2d(64, eps=1e-05),
    nn.Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False),
    nn.BatchNorm2d(256, eps=1e-05),
    nn.ReLU(inplace=True)
)
```

Output Layer Parameters: output layer is a convolution 2d layer with the parameters shown in the below image.

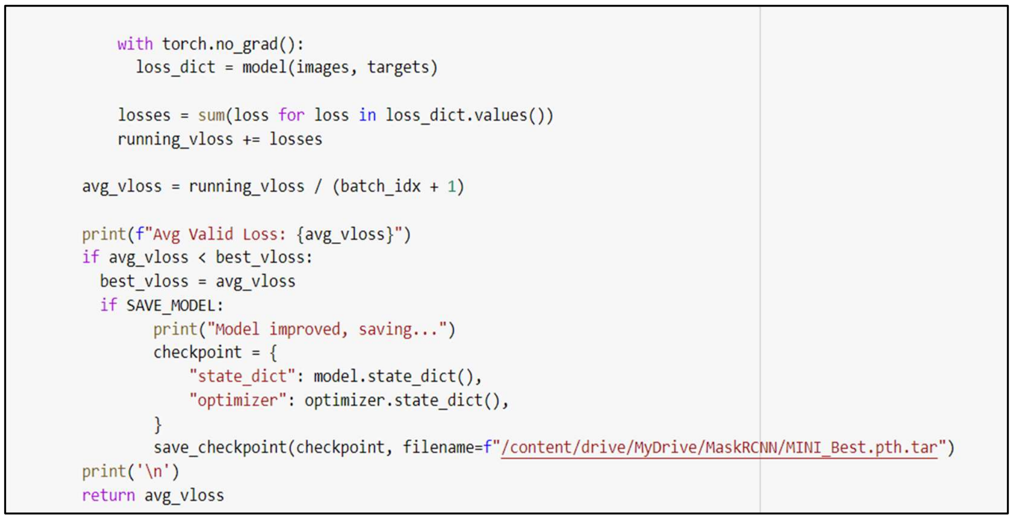


```
      )
      (mask_predictor): MaskRCNNPredictor(
        (conv5_mask): ConvTranspose2d(256, 256, kernel_size=(2, 2), stride=(2, 2))
        (relu): ReLU(inplace=True)
        (mask_fcn_logits): Conv2d(256, 91, kernel_size=(1, 1), stride=(1, 1))
      )
    )
  )
```

Experiments and Results:

Fine Tuning: This model is already fine-tuned as the backbone layer is changed.

Model Selection: Loss is calculated at each step and at minimum loss the model state of dict is saved as the best model.

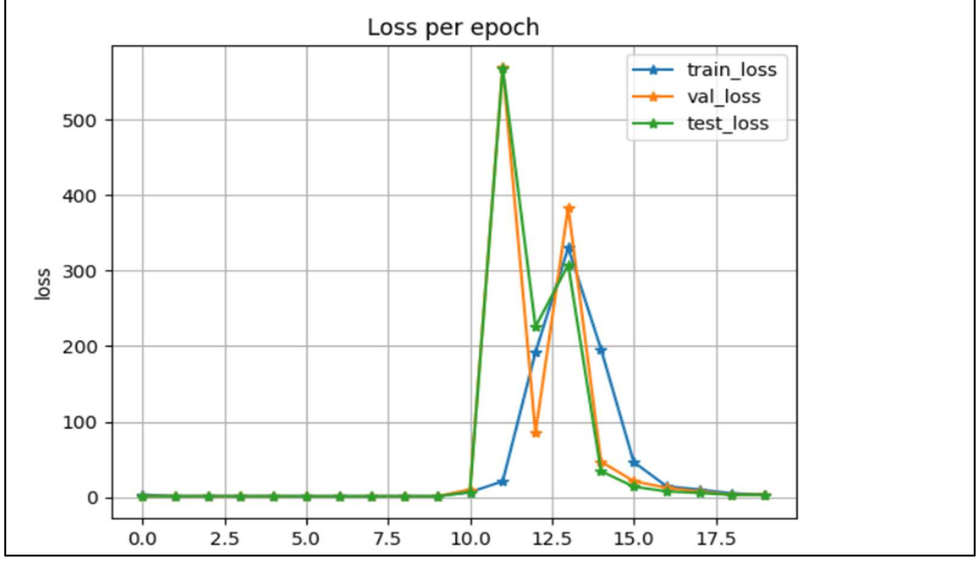


```
    with torch.no_grad():
      loss_dict = model(images, targets)

    losses = sum(loss for loss in loss_dict.values())
    running_vloss += losses

avg_vloss = running_vloss / (batch_idx + 1)

print(f"Avg Valid Loss: {avg_vloss}")
if avg_vloss < best_vloss:
  best_vloss = avg_vloss
  if SAVE_MODEL:
        print("Model improved, saving...")
        checkpoint = {
            "state_dict": model.state_dict(),
            "optimizer": optimizer.state_dict(),
        }
        save_checkpoint(checkpoint, filename=f"/content/drive/MyDrive/MaskRCNN/MINI_Best.pth.tar")
print('\n')
return avg_vloss
```

Best Model Selection

Evaluation: Losses for the train dataset, validation dataset, and test dataset are calculated to observe the model performance.

```
plt.plot([i.item() for i in train_loss], label='train_loss', marker='*')
plt.plot([i.item() for i in val_loss], label='val_loss', marker='*')
plt.plot([i.item() for i in test_loss], label='test_loss', marker='*')
plt.title('Loss per epoch'); plt.ylabel('loss');
plt.xlabel('epoch')
plt.legend(), plt.grid()
plt.show()
```



Conclusion

Overall, we manually assembled the dataset and turned it into a custom dataset. We first attempted transfer learning with the custom data using the MaskRCNN pretrained model, and obtained an accuracy of 51%. We created a small network with four successive layers to further boost speed. This network served as the backbone layer for the MaskRCNN. The model loss was decreased to a very low amount using this method, which greatly reduced it. Combining these methods, we were able to distinguish and precisely identify distinct instances within the dataset, achieving instance segmentation.

References

The reference for this project is being taken from multiple papers dedicated and more papers will be studied in the same regards in the coming future.

1. He, K., Gkioxari, G., Dollár, P. and Girshick, R., 2017. Mask r-cnn. In Proceedings of the IEEE international conference on computer vision (pp. 2961-2969). https://arxiv.org/pdf/1703.06870.pdf

2. Zhou, C., 2020. Yolact++ Better Real-Time Instance Segmentation. University of California, Davis. (https://ieeexplore-ieee-org.unh-proxy01.newhaven.edu/document/9159935)

Reference Code

https://debuggercafe.com/instance-segmentation-with-pytorch-and-mask-r-cnn/