# CS 498: Assignment 4: Segmentation

Due on April 8, 2022

March 25, 2022

## Submission

In this assignment, you will implement semantic segmentation using neural networks. The starter code consists of an iPython notebook "mp4.ipynb" which can be opened and run on Google colab. Please put together a single PDF with your answers and figures for each problem, and submit it to Gradescope (Course Code: JBXJVZ). We recommend you add your answers to the latex template files we provided. More details on what to report are in the provided notebook.

Reminder: please put your name and netid in your pdf. Your submission should include your pdf and filled out mp4.ipynb.

## Semantic Segmentation

**Question 1 (Data loading and augmentation)[1 pt]:** We provide code that loads the segmentation data. In this part you will need to perform data augmentation on the loaded data within the "SegmentationDataset" class. In particular you should take a random crop of the image and with some probability you should flip the image horizontally. You should experiment with different probabilities and crop sizes and report the results in your pdf. Make sure to use pytorch built in transforms methods.

```
if self.transform:
#--------------------------
# apply random resized crop
i, j, h, w = transforms.RandomResizedCrop.get_params(img, scale=(0.8,1), ratio=(3/4,4/3))
img = TF.resized_crop(img, i, j, h, w, (img.size[1], img.size[0]))
gt = TF.resized_crop(gt, i, j, h, w, (gt.size[1], gt.size[0]))

# apply random horizontal flip
if random.random() < 0.5:
    img = TF.hflip(img)
    gt = TF.hflip(gt)
#--------------------------
```

**Question 2 (Simple Baseline) [2 pts]:** In this part you will be modifying "simple_train" and "simple_predict". For each pixel you should compute the distribution of class labels at that pixel from the training dataset. When predicting classes for a new image, simply output these class frequencies at each pixel. You can run the evaluation code (from the next question) with your simple baseline and see its mean average precision which we provide - it should be around 24.

```
# Question 2
# Output shape: (num_classes, 224, 288)
def simple_train(num_classes, train_dataloader):
    #--------------------------
```


(a) default (b) scaling (c) aspect ratio (d) flipping

Figure 1: Comparisons of image samples when (a) no transform, (b) scaling, (c) change in aspect ratio, and (d) horizontal flipping is applied.

```
    model = torch.zeros(num_classes, 224, 288)

    for i, (img, gt) in enumerate(dataloader):
        for class_label in range(num_classes):
            model[class_label] += (gt[:, 0] == class_label).sum(dim=0)

    # normalize
    model /= model[:,0,0].sum()
    model = model.numpy()
    # --------------------------------
    return model

# Output:
#   gt: the ground truth segmentation, shape (B, 1, 224, 288)
#   preds: the predicted segmentation class probabilities, shape (B, 9, 224, 288)
def simple_predict(dataloader, model):
    gts, preds = [], []
    for i, batch in enumerate(dataloader):
        # --------------------------------
        img_batch, gt_batch = batch

        for j in range(len(batch)):
            gts.append(gt_batch[j].numpy())
            preds.append(model)

        # --------------------------------
    return np.array(gts), np.array(preds), list(dataset.classes)
```

**Question 3 (Evaluation Metrics) [1 pts]:** We must evaluate the quality of our predictions. In this part you will fill in "compute_confusion_matrix". You should write code to compute the confusion matrix as well as IoU for the predicted segmentation when compared to ground truth. We provide code for

Figure 2: Results of the simple prediction algorithm.

visualizing the computed values as well as computing mean average precision.

```
# Question 3: compute the confusion matrix and IoU metrics
# Hint: once you've computed the confusion matrix, IoU is easy
# Note: preds contains class probabilities, convert this to a class prediction
def compute_confusion_matrix(gts, preds):
    # ----------------------------
    n_classes = 9
    conf = np.zeros((n_classes, n_classes))
    IoU = np.zeros((n_classes))

    gts_argmax = gts[:,0]
    preds_argmax = np.argmax(preds, axis=1)

    for i in range(n_classes):
        for j in range(n_classes):
            conf[i][j] = np.count_nonzero((gts_argmax==i)*(preds_argmax==j))

    IoU = conf.diagonal() / conf.sum(axis=1)
    # ----------------------------
    return IoU, conf
```

**Question 4 (Loss function) [2 pt]:** To train a model we need a loss function. In this part you will fill in "cross_entropy_criterion" with your implementation of the weighted cross entropy between predicted class probabilities and ground truth class labels.

```
def cross_entropy_criterion(predictions, labels, weights, device):
    # https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html#torch.nn.CrossEntropyLoss
    # https://gist.github.com/yang-zhang/217dcc6ae9171d7a46ce42e215c1fee0

    B, C, H, W = predictions.shape

    if type(predictions) is not torch.Tensor:
        predictions = torch.from_numpy(predictions)
    if type(labels) is not torch.Tensor:
        labels = torch.from_numpy(labels)

    pred = predictions.reshape(B,C,-1).to(device)
    target = labels.reshape(B,-1).to(device)

    pred = F.log_softmax(pred, dim=1)
```

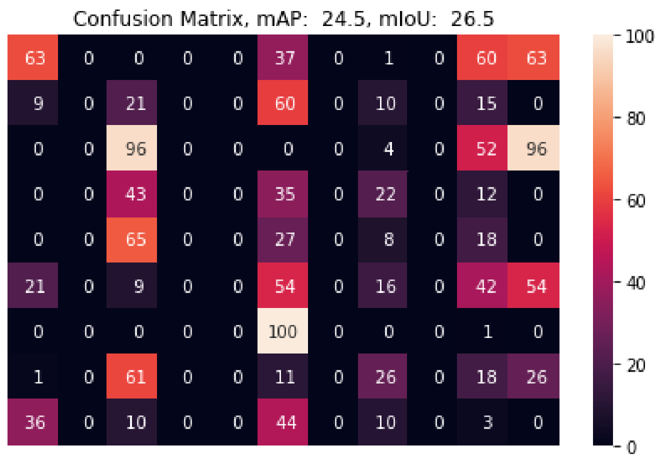


Figure 3: Evaluation on the results of the simple prediction.

```
    loss = F.nll_loss(pred, target, torch.FloatTensor(weights).to(device))

    return loss
```

**Question 5 (Train loop) [2 pt]:** In this part you will implement the stochastic gradient descent training loop in pytorch, modifying ”train”. We provide code to validate a trained model and a skeleton for training one.

```
def train(model, optimizer, criterion, trainloader, device,
          scheduler=None, epochs=100, num_early_stop=np.nan,
          valloader=None, save_path=MODEL_PATH):
    # ---------------------------------
    best_model_fname = "model_best.pth"
    best_model_path = os.path.join(save_path, best_model_fname)

    # create a directory to save checkpoints
    if not os.path.exists(save_path):
        os.makedirs(save_path)
        print("Create %s" % (save_path))
    # ---------------------------------
    ...
    for epoch in range(epochs):
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # ---------------------------------
            img, gt = data
            img = img.to(device)
            gt = gt.to(device)

            optimizer.zero_grad()
            pred = model(img)
            loss = criterion(pred, gt).to(device)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            # ---------------------------------
        ...

        if not valloader is None:
            val_loss = 0.0
            with torch.no_grad():
                for data in valloader:
                    # ---------------------------------
```

```
            img, gt = data
            img = img.to(device)
            gt = gt.to(device)

            pred = model(img)
            loss = criterion(pred, gt)
            val_loss += loss.item()
            # ---------------------------
        ...
        # ---------------------------
        # step over scheduler
        if not scheduler is None:
            scheduler.step(np.mean(val_loss))

        # check whether the performance has been improved. if not, quit training
        if val_loss == min(val_loss_over_epochs):
            torch.save(model.state_dict(), best_model_path)
            print(f"Epoch: {epoch+1}, Best model saved at {best_model_path}")
            num_no_improvement = 0
        else:
            num_no_improvement += 1
            print(f"Epoch: {epoch+1}, No improvement ({num_no_improvement} / {num_early_stop})")
            if num_no_improvement >= num_early_stop:
                break
        # ---------------------------
    ...
    # ---------------------------
    plt.tight_layout()
    plt.savefig(os.path.join(save_path, "loss_curve.png"))
    # ---------------------------
    return model
```

**Question 6 (Model definition and training) [4 pt]:** Implement a basic convolutional neural network, as well as the U-Net architecture for semantic segmentation. Train your models with the code you wrote for Question 5.

```
# credit from https://github.com/ptrblck/pytorch_misc/blob/master/unet_demo.py

# basic CNN block consisting of convnet + activation + batch normalization
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels,
                 kernel_size=3, padding=1, stride=1):
        super(ConvBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels,
                              kernel_size=kernel_size, padding=padding,
                              stride=stride, bias=False)
        self.batch_norm = nn.BatchNorm2d(out_channels, eps=1e-05, momentum=0.1,
                                         affine=True, track_running_stats=True)
        self.activation = nn.ReLU(inplace=True)

    def forward(self, x):
        x = self.conv(x)
        x = self.activation(x)
        x = self.batch_norm(x)

        return x

# CNN block for downsampling consisting of BaseConv followed by pooling
# (i.e., reduce tensor side length by half)
class DownConv(nn.Module):
    def __init__(self, in_channels, out_channels,
                 kernel_size=3, padding=1, stride=1):
        super(DownConv, self).__init__()
        self.conv_act_bn = ConvBlock(in_channels, out_channels,
                                     kernel_size, padding, stride)
        self.pooling = nn.MaxPool2d(kernel_size=3, padding=1, stride=2,
```

```
                                   dilation=1, ceil_mode=False)

    def forward(self, x):
        x = self.pooling(x)      # reduce side of tensor by half
        x = self.conv_act_bn(x)

        return x

# CNN block for upsampling consisting of ConvTranspose2d followed by BaseConv
# (i.e., increase tensor side length twice)
class UpConv(nn.Module):
    def __init__(self, in_channels, out_channels,
                 kernel_size=3, padding=1, stride=1):
        super(UpConv, self).__init__()
        self.conv_act_bn = ConvBlock(in_channels, out_channels,
                                     kernel_size, padding, stride)
        self.conv_transpose = nn.ConvTranspose2d(in_channels, in_channels,
                                                 kernel_size=2, padding=0,
                                                 stride=2, bias=False)

    def forward(self, x):
        x = self.conv_transpose(x)  # increase side of tensor twice
        x = self.conv_act_bn(x)

        return x

# CNN block for upsampling with skip connection in UNet architecture,
# consisting of ConvTranspose2d followed by BaseConv
# (i.e., increase tensor side length twice)
class UnetUpConv(nn.Module):
    def __init__(self, in_channels, in_channels_skip, out_channels,
                 kernel_size=3, padding=1, stride=1):
        super(UnetUpConv, self).__init__()
        self.conv_act_bn = ConvBlock(in_channels+in_channels_skip, out_channels,
                                     kernel_size, padding, stride)
        self.conv_transpose = nn.ConvTranspose2d(in_channels, in_channels,
                                                 kernel_size=2, padding=0,
                                                 stride=2, bias=False)

    def forward(self, x, x_skip):
        x = self.conv_transpose(x)          # increase side of tensor twice
        x = torch.cat((x, x_skip), dim=1)   # concatenate upsampled and skipped tensor
        x = self.conv_act_bn(x)             # proceed to convnet operation

        return x

class BaseConv(nn.Module):
def __init__(self, n_classes=9):
    super(BaseConv, self).__init__()

    in_channels = 3
    out_channels = 64

    # block for convnet operation for input tensor
    self.conv_in = ConvBlock(in_channels=in_channels, out_channels=out_channels,
                             kernel_size=7, padding=3, stride=2)

    # blocks for convnet operation followed by downsampling
    self.conv_down1 = DownConv(in_channels=1*out_channels, out_channels=1*out_channels)
    self.conv_down2 = DownConv(in_channels=1*out_channels, out_channels=2*out_channels)
    self.conv_down3 = DownConv(in_channels=2*out_channels, out_channels=4*out_channels)
    self.conv_down4 = DownConv(in_channels=4*out_channels, out_channels=8*out_channels)

    # blocks for convnet operation preceded by upsampling
    self.conv_up4 = UpConv(in_channels=8*out_channels, out_channels=4*out_channels)
    self.conv_up3 = UpConv(in_channels=4*out_channels, out_channels=2*out_channels)
    self.conv_up2 = UpConv(in_channels=2*out_channels, out_channels=1*out_channels)
    self.conv_up1 = UpConv(in_channels=1*out_channels, out_channels=1*out_channels)
```

```python
        self.conv_up0 = UpConv(in_channels=1*out_channels, out_channels=1*out_channels)

        # block for convnet operation to construct n_classes semantic outputs
        self.conv_out = nn.Conv2d(in_channels=out_channels, out_channels=n_classes,
                                  kernel_size=1, padding=0, stride=1)

    def __repr__(self):
        return "BaseConv"

    def forward(self, x):
        x = self.conv_in(x)

        x_down1 = self.conv_down1(x)
        x_down2 = self.conv_down2(x_down1)
        x_down3 = self.conv_down3(x_down2)
        x_down4 = self.conv_down4(x_down3)

        x_up4 = self.conv_up4(x_down4)
        x_up3 = self.conv_up3(x_down3)
        x_up2 = self.conv_up2(x_up3)
        x_up1 = self.conv_up1(x_up2)
        x_up0 = self.conv_up1(x_up1)

        pred = self.conv_out(x_up0)

        return pred

class UNet(nn.Module):
    def __init__(self, n_classes=9):
        super(UNet, self).__init__()

        in_channels = 3
        out_channels = 64

        # block for convnet operation for input tensor
        self.conv_in = ConvBlock(in_channels=in_channels, out_channels=out_channels,
                                 kernel_size=7, padding=3, stride=2)

        # blocks for convnet operation followed by downsampling
        self.conv_down1 = DownConv(in_channels=1*out_channels, out_channels=1*out_channels)
        self.conv_down2 = DownConv(in_channels=1*out_channels, out_channels=2*out_channels)
        self.conv_down3 = DownConv(in_channels=2*out_channels, out_channels=4*out_channels)
        self.conv_down4 = DownConv(in_channels=4*out_channels, out_channels=8*out_channels)

        # blocks for convnet operation preceded by upsampling
        self.conv_up4 = UnetUpConv(in_channels=8*out_channels, in_channels_skip=4*out_channels,
                                   out_channels=4*out_channels)
        self.conv_up3 = UnetUpConv(in_channels=4*out_channels, in_channels_skip=2*out_channels,
                                   out_channels=2*out_channels)
        self.conv_up2 = UnetUpConv(in_channels=2*out_channels, in_channels_skip=1*out_channels,
                                   out_channels=1*out_channels)
        self.conv_up1 = UnetUpConv(in_channels=1*out_channels, in_channels_skip=1*out_channels,
                                   out_channels=1*out_channels)
        self.conv_up0 = UpConv(in_channels=1*out_channels, out_channels=1*out_channels)

        # block for convnet operation to construct n_classes semantic outputs
        self.conv_out = nn.Conv2d(in_channels=out_channels, out_channels=n_classes,
                                  kernel_size=1, padding=0, stride=1)

    def __repr__(self):
        return "UNet"

    def forward(self, x):
        x = self.conv_in(x)
        x_down1 = self.conv_down1(x)
        x_down2 = self.conv_down2(x_down1)
        x_down3 = self.conv_down3(x_down2)
        x_down4 = self.conv_down4(x_down3)
```

```
        x_up4 = self.conv_up4(x_down4, x_down3)
        x_up3 = self.conv_up3(x_up4, x_down2)
        x_up2 = self.conv_up2(x_up3, x_down1)
        x_up1 = self.conv_up1(x_up2, x)
        x_up0 = self.conv_up0(x_up1)

        pred = self.conv_out(x_up0)

        return pred

# First make the model and put it on the device
base_model = BaseConv().to(device)

# Now define our loss criterion as cross entropy based on your previous code
criterion = lambda y_pred, y_true: cross_entropy_criterion(y_pred, y_true, class_weights, device)

# Now make our optimizer for this model
optimizer = optim.SGD(base_model.parameters(), lr=1e-3, momentum=0.9, weight_decay=0.01)

# Now train and validate
base_model = train(base_model, optimizer, criterion, dataloader, device,
                   valloader=val_dataloader, save_path=os.path.join(MODEL_PATH, str(base_model)),
                   epochs=50, scheduler=None, num_early_stop=np.nan)
preds, gts = validate_model(val_dataloader, base_model, list(dataset.classes), device)
```

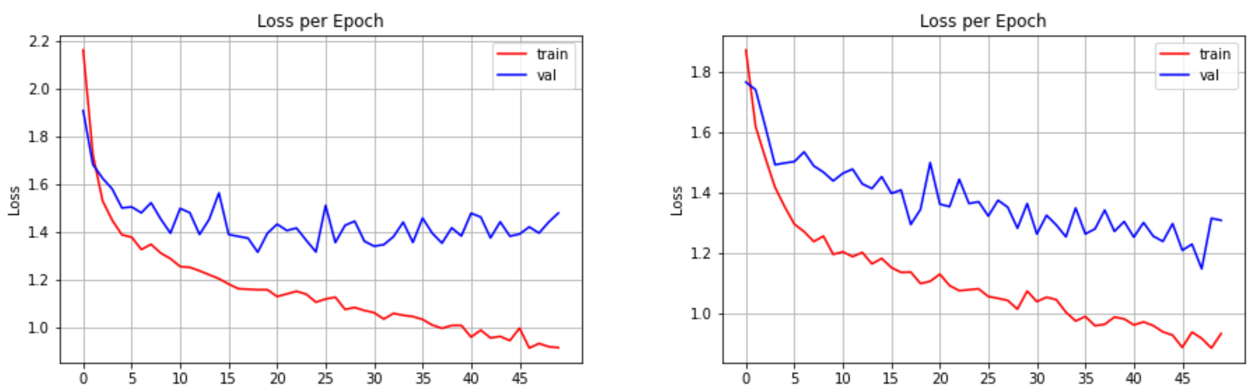


Figure 4: Loss curve of training `BaseConv` (left) and `UNet` (right) models, respectively.

```
# BaseConv
sky:  AP: 0.95, IoU: 0.90
tree:  AP: 0.62, IoU: 0.65
road:  AP: 0.90, IoU: 0.78
grass:  AP: 0.59, IoU: 0.71
water:  AP: 0.18, IoU: 0.45
building:  AP: 0.68, IoU: 0.55
mountain:  AP: 0.04, IoU: 0.28
foreground:  AP: 0.59, IoU: 0.56
misc:  AP: 0.01, IoU: 0.06
mean:  AP: 0.50, IoU: 0.55
# UNet
sky:  AP: 0.95, IoU: 0.87
tree:  AP: 0.74, IoU: 0.75
road:  AP: 0.93, IoU: 0.81
grass:  AP: 0.72, IoU: 0.77
```

```
water:  AP: 0.29, IoU: 0.57
building:  AP: 0.74, IoU: 0.65
mountain:  AP: 0.06, IoU: 0.27
foreground:  AP: 0.68, IoU: 0.65
misc:  AP: 0.01, IoU: 0.09
mean:  AP: 0.57, IoU: 0.60
```

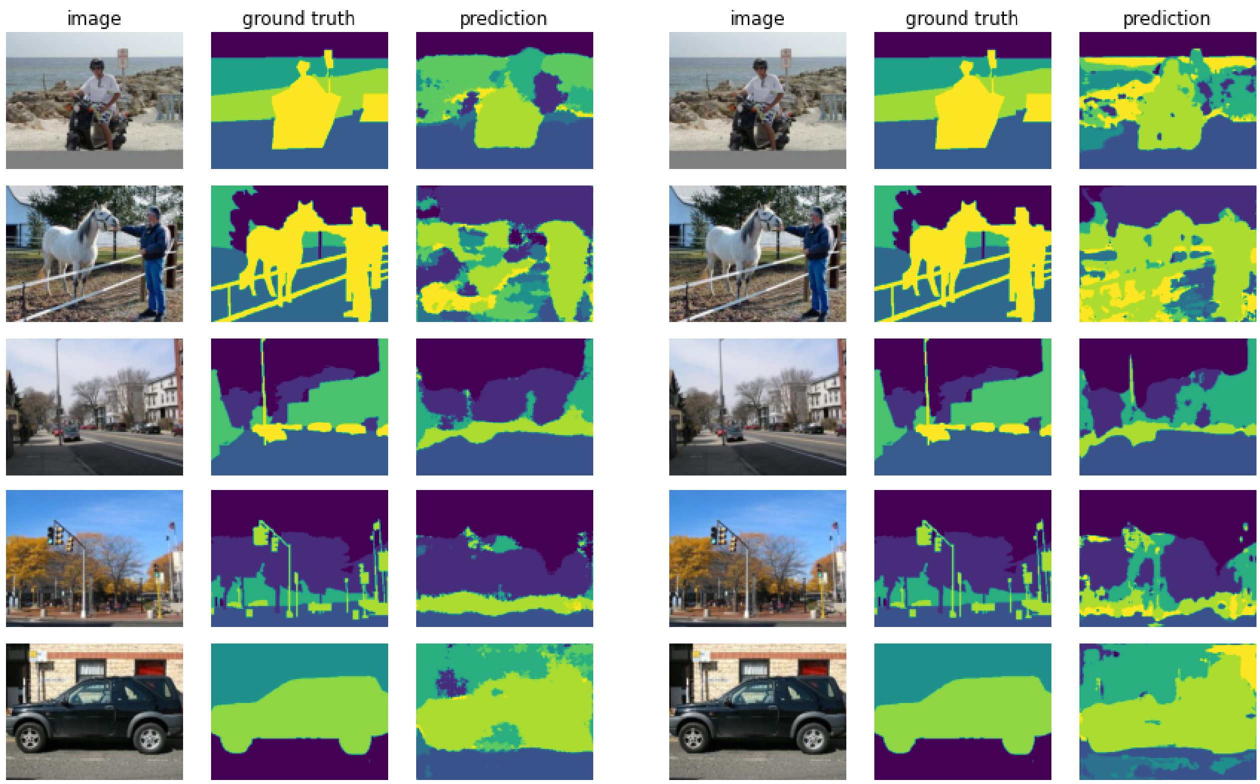


Figure 5: Predictions made by `BaseConv` (left) and `UNet` (right) models, respectively.

**Question 7 (Use Pretrained Model) [3 pt]:** In this part you will build on resnet-18 (note there are multiple ways to do this). Report your results, they should be better than the best you got using UNet training from scratch.

```
class ResnetBasedModel(nn.Module):
def __init__(self, pretrained_resnet, num_layers_to_remove=0, n_classes=9):
    super(ResnetBasedModel, self).__init__()
    # You can, for example, extract the first N layers of the model like this:
    # self.resnet_features = nn.Sequential(*list(pretrained_resnet.children())[:N])

    out_channels = 64

    resnet_layers = list(pretrained_resnet.children())

    # blocks for convnet operation followed by downsampling
    self.conv_in = nn.Sequential(*resnet_layers[:3])
    self.conv_down1 = nn.Sequential(*resnet_layers[3:5])
    self.conv_down2 = resnet_layers[5]
    self.conv_down3 = resnet_layers[6]
    self.conv_down4 = resnet_layers[7]
```

```
        # blocks for convnet operation preceded by upsampling, without skip connection
        self.conv_up4 = UpConv(in_channels=8*out_channels, out_channels=4*out_channels)
        self.conv_up3 = UpConv(in_channels=4*out_channels, out_channels=2*out_channels)
        self.conv_up2 = UpConv(in_channels=2*out_channels, out_channels=1*out_channels)
        self.conv_up1 = UpConv(in_channels=1*out_channels, out_channels=1*out_channels)
        self.conv_up0 = UpConv(in_channels=1*out_channels, out_channels=1*out_channels)

        # block for convnet operation to construct n_classes semantic outputs
        self.conv_out = nn.Conv2d(in_channels=out_channels, out_channels=n_classes,
                                  kernel_size=1, padding=0, stride=1)

    def __repr__(self):
        return "ResnetBasedModel"

    def forward(self, x):
        x = self.conv_in(x)
        x_down1 = self.conv_down1(x)
        x_down2 = self.conv_down2(x_down1)
        x_down3 = self.conv_down3(x_down2)
        x_down4 = self.conv_down4(x_down3)

        x_up4 = self.conv_up4(x_down4)
        x_up3 = self.conv_up3(x_up4)
        x_up2 = self.conv_up2(x_up3)
        x_up1 = self.conv_up1(x_up2)
        x_up0 = self.conv_up0(x_up1)

        pred = self.conv_out(x_up0)

        return pred

        class ResnetUNetModel(nn.Module):
        def __init__(self, pretrained_resnet, num_layers_to_remove=0, n_classes=9):
            super(ResnetUNetModel, self).__init__()
            # You can, for example, extract the first N layers of the model like this:
            # self.resnet_features = nn.Sequential(*list(pretrained_resnet.children())[:N])

            out_channels = 64

            resnet_layers = list(pretrained_resnet.children())

            # blocks for convnet operation followed by downsampling
            self.conv_in = nn.Sequential(*resnet_layers[:3])
            self.conv_down1 = nn.Sequential(*resnet_layers[3:5])
            self.conv_down2 = resnet_layers[5]
            self.conv_down3 = resnet_layers[6]
            self.conv_down4 = resnet_layers[7]

            # blocks for convnet operation preceded by upsampling, with skip connection
            self.conv_up4 = UnetUpConv(in_channels=8*out_channels, in_channels_skip=4*out_channels,
                                       out_channels=4*out_channels)
            self.conv_up3 = UnetUpConv(in_channels=4*out_channels, in_channels_skip=2*out_channels,
                                       out_channels=2*out_channels)
            self.conv_up2 = UnetUpConv(in_channels=2*out_channels, in_channels_skip=1*out_channels,
                                       out_channels=1*out_channels)
            self.conv_up1 = UnetUpConv(in_channels=1*out_channels, in_channels_skip=1*out_channels,
                                       out_channels=1*out_channels)
            self.conv_up0 = UpConv(in_channels=1*out_channels, out_channels=1*out_channels)

            # block for convnet operation to construct n_classes semantic outputs
            self.conv_out = nn.Conv2d(in_channels=out_channels, out_channels=n_classes,
                                      kernel_size=1, padding=0, stride=1)

        def __repr__(self):
            return "ResnetUNetModel"

        def forward(self, x):
            x = self.conv_in(x)
```

```
        x_down1 = self.conv_down1(x)
        x_down2 = self.conv_down2(x_down1)
        x_down3 = self.conv_down3(x_down2)
        x_down4 = self.conv_down4(x_down3)

        x_up4 = self.conv_up4(x_down4, x_down3)
        x_up3 = self.conv_up3(x_up4, x_down2)
        x_up2 = self.conv_up2(x_up3, x_down1)
        x_up1 = self.conv_up1(x_up2, x)
        x_up0 = self.conv_up0(x_up1)

        pred = self.conv_out(x_up0)

        return pred
```

```
# ResnetBasedModel
sky:  AP: 0.94, IoU: 0.84
tree:  AP: 0.75, IoU: 0.80
road:  AP: 0.94, IoU: 0.88
grass:  AP: 0.36, IoU: 0.50
water:  AP: 0.76, IoU: 0.80
building:  AP: 0.87, IoU: 0.76
mountain:  AP: 0.16, IoU: 0.31
foreground:  AP: 0.78, IoU: 0.70
misc:  AP: 0.04, IoU: 0.42
mean:  AP: 0.62, IoU: 0.67
# ResnetUNetModel
sky:  AP: 0.97, IoU: 0.91
tree:  AP: 0.81, IoU: 0.77
road:  AP: 0.95, IoU: 0.86
grass:  AP: 0.59, IoU: 0.65
water:  AP: 0.67, IoU: 0.78
building:  AP: 0.88, IoU: 0.80
mountain:  AP: 0.10, IoU: 0.60
foreground:  AP: 0.84, IoU: 0.81
misc:  AP: 0.00, IoU: 0.09
mean:  AP: 0.65, IoU: 0.70
```

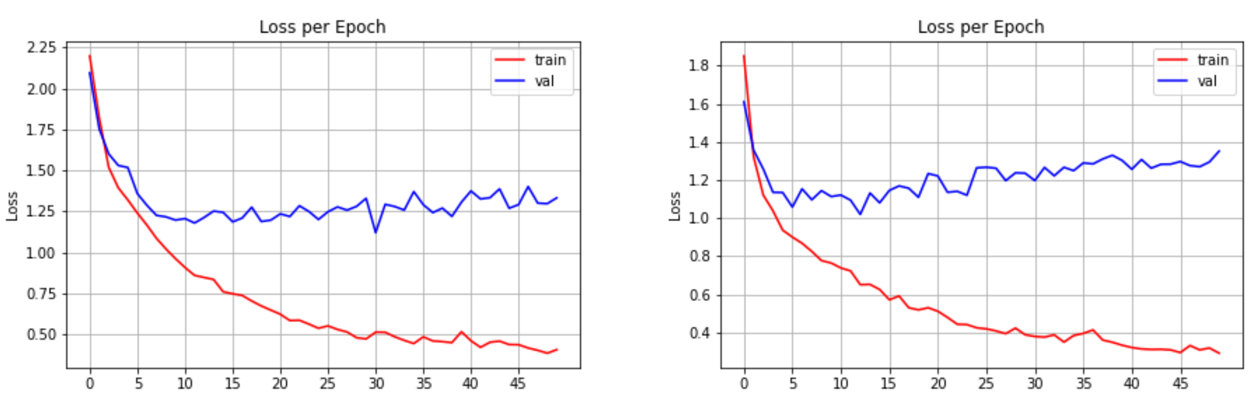


Figure 6: Loss curve of training ResNet-based models without skip connection (left) and with skip connection (right), respectively.

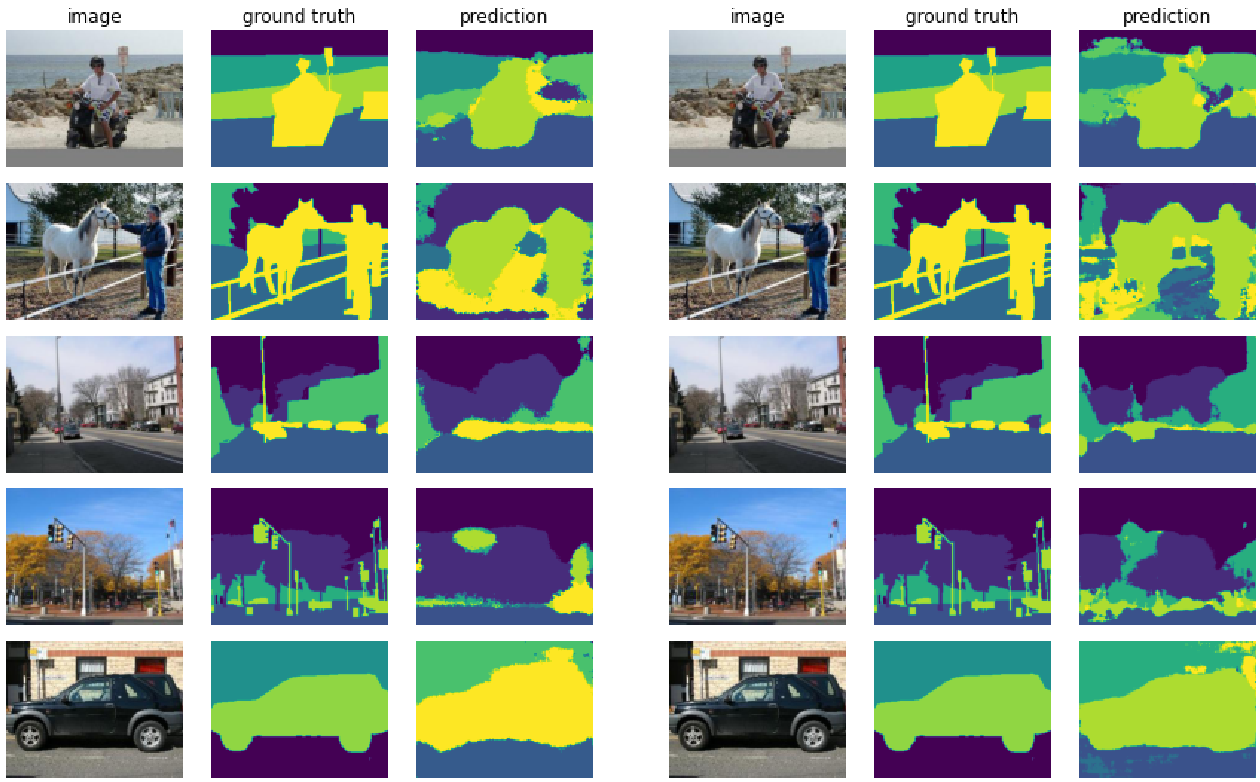


Figure 7: Predictions made by ResNet-based models without skip connection (left) and with skip connection (right), respectively.

## Instance Segmentation (Bonus 4pt)

Now we have a deep semantic segmentation algorithm. However, the model cannot distinguish each instance. Could you use a similar UNet model to build an instance segmentation algorithm? Please download the Upenn-Fudan pedestrian dataset here `https://www.cis.upenn.edu/~jshi/ped_html/` and get their instance labels. There are two types of instance segmentation methods, detection-free or detection-based. Choose either one of them.

Please refer to the Deep Watershed Transform for an example of detection-free method `https://github.com/min2209/dwt`. Your goal is to build a network with two headers, one to predict the binary semantic label similar to your semantic segmentation network and the other to predict the distance transform to the boundary. Once you have these two, the watershed transform could be applied to recover per-pixel instance labels.

For the detection-based method, please refer to the MaskRCNN (`https://pytorch.org/vision/stable/_modules/torchvision/models/detection/mask_rcnn.html`). You will develop a network that predicts detection proposals first. Then within each detection proposal, a binary foreground and background segmentation is conducted to separate each instance.

For each method, you should implement 1) the loss function the paper uses; 2) implement the data loader and post-processing that converts network output to per-pixel instance label 3) train and evaluation each model (you could either reuse your UNet backbone or follow the original paper).