CS 498: Assignment 4: Segmentation

Chenhui Zhang (chenhui5)

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.

Answer: In my implementation for data augmentation, I created custom transformations with `torchvision`'s functional interface to apply the same same transformation to both image and ground truth. For horizontal flip, I generate a random number from 0 to 1 each time and compare it with parameter `p`, the probability of flip. If the random number is smaller than `p`, I apply flip to both the image and the mask. For random crop, I used `RandomResizedCrop` to crop the image and the mask to a smaller size and resize them back to the original size. I get the random crop and resize parameters for the image and apply the same parameters to the mask with `torchvision`'s functional interface. At validation or test time, I do not apply random flip or crop.

For the flip probability, I simply chose 0.5 because I don't want the training data to be skewed towards any direction since the purpose of introducing this kind of data augmentation is to help the model to learn features that are invariant to flipping. For the random resized crop, I chose the scale ratio of the cropped region w.r.t. the original image to be between 0.75 and used the default bounds for aspect ratio (0.75, 1.333). This set of crop parameters is chosen to enable the model to learn the invariance against cropping and resizing to some degree while preventing the cropped patch to be too localized.

Since this Piazza post says we can "just pick one [set of parameters] and justify why it is reasonable," I do not attach comparison results here.

Answer for Notebook Question: Please see the visualization of image transformations in the next page.

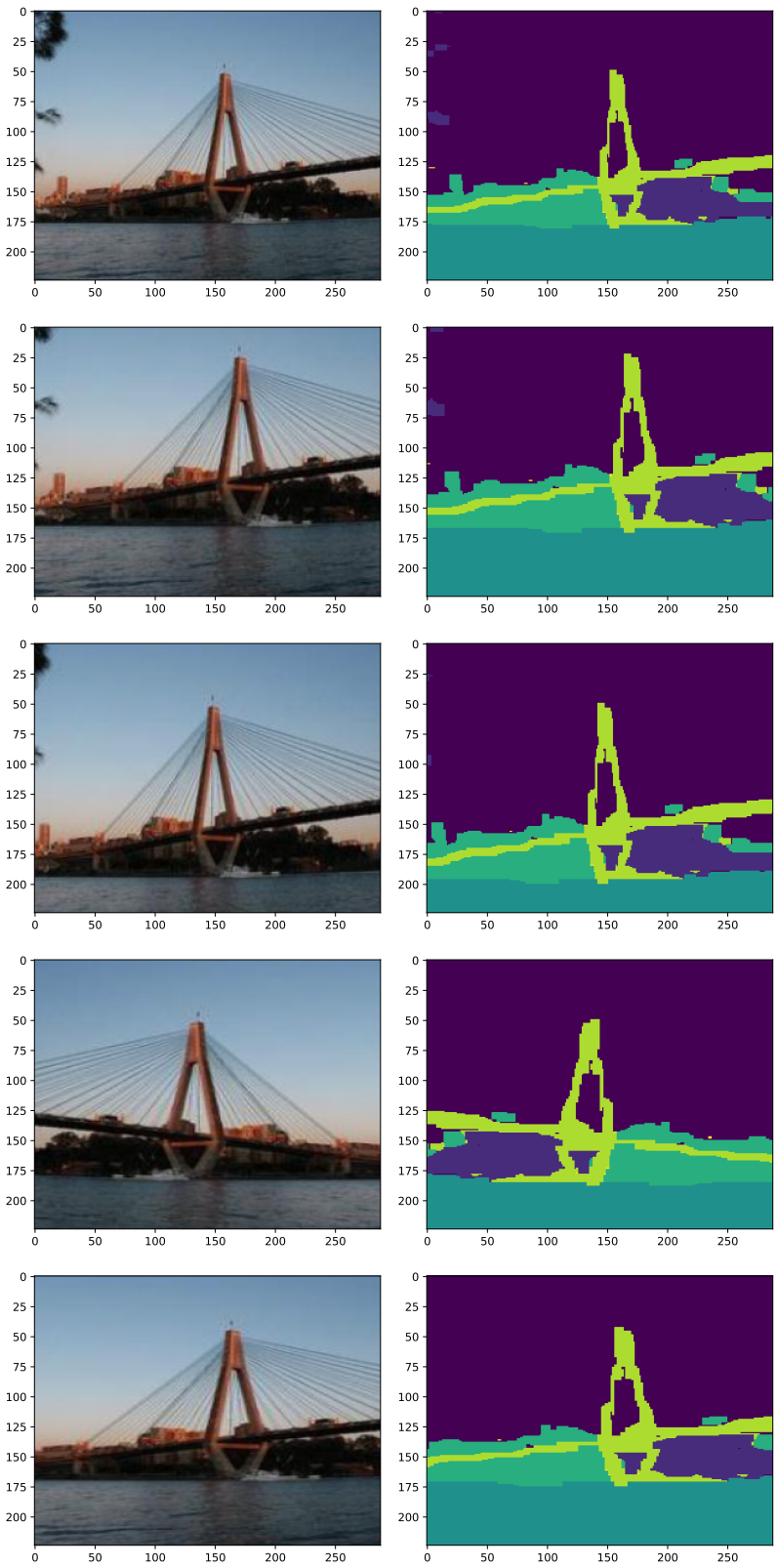
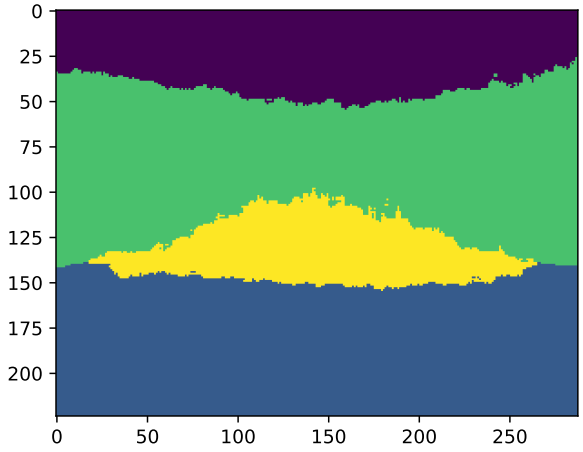


Figure 1: Image Transformations

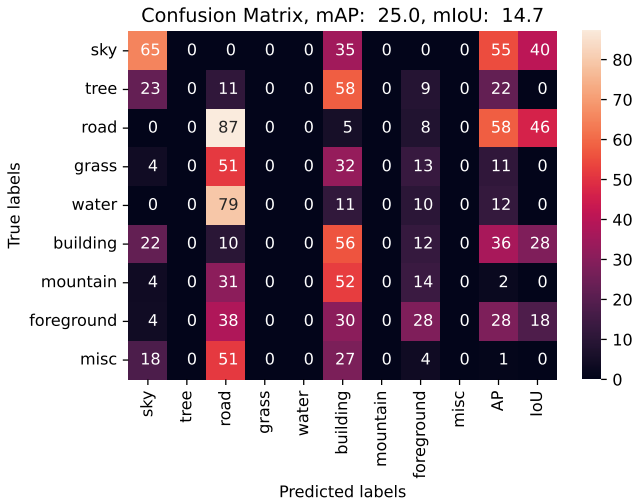
Question 2 (Simple Baseline) [2 pts]: In this part you will be modifying "`simple_train`" and "`simple_predict`". For each pixel you should compute the distribution of class labels at that pixel from the training dataset. When predicting classes for a new image, simply output these class frequencies 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.

Answer: For `simple_train`, I simply count the number of class labels at each pixel and normalize the count to probabilities. For `simple_predict`, I just predict the same probabilities calculated in `simple_train`.

Answer for Notebook Question: Please check the evaluation metrics and output image below.



(a) Simple Prediction



(b) Confusion Matrix

Figure 2: Simple Baseline Result

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

Answer: To calculate confusion matrix, I first take the `argmax` across the prediction logits to calculate the predicted label for each class at each pixel. Then, I flatten the prediction tensor and the ground truth tensor. Note that the arrary of (1, `y_pred`, `y_true`) tuples naturally defines the confusion matrix in COO format. I utilized this property to construct the confusion matrix without any loop. Then, the intersection of prediction and ground truth classes is just the diagonal of the confusion matrix. To get the union of prediction and ground truth classes, we add up the row sum and the column sum of the confusion matrix and subtract the diagonal (intersection). Finally, we have the IoU for each class. I compared the results to sklean's `confusion_matrix` and `jaccard_score`, and the results are the same.

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.

Answer: To implement cross entropy loss, I first obtained the log probability scores from raw logits for every class at each pixel by using `F.log_softmax`. Then, I broadcasted the weight vector to the shape of labels $(N, 1, H, W)$ according to the label at each pixel. Then, I selected the log probability scores at

label indices. Note that labels are $(N, 1, H, W)$ and log probabilities are $(N, 9, H, W)$ so we can select the log probabilities with indices provided by the labels by `torch.gather`.

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_{y_n} x_{n,y_n}, \quad w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore\_index}\},$$

where x is the input, y is the target, w is the weight, and N is the batch size.

Finally, I take the weighted average of the selected negative log probabilies to get the cross entropy loss.

$$\ell(x, y) = \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n}} l_n$$

I compared my implementation with `PyTorch` and the results are the same.

Answer for Notebook Questions: Cross entropy loss is the cross entropy between the empirical label distribution and the "estimated" class probability distribution from our model. The detailed formulation and implementation is described above. The purpose of weighting is to account for imbalanced classes. For example, image background (stuff) could have a huge number of pixels while well-defined objects (things) could have smaller number of pixels. Without weighting, performing well simply on background could yield small loss. Therefore, we want to give more weight in the loss to rare classes so that they are not ignored by our model.

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.

Answer: For the training loop, I first switch the model to training mode, send the tensors to GPU, and calculate the model prediction. Then, I calculate the loss function and take gradients w.r.t. model parameters. Finally, I update the model parameters with optimizers and update the running loss. At each step, I also clear the gradients of the optimizer so that parameters are updated correctly. For evaluation loop, the logic is similar to training but we don't calculate gradient or update model parameters. I also switch the model to evaluation mode so that Batch Normalization could work properly.

Answer for Notebook Questions: It is important to consider both validation and training losses simultaneously because we would like our model to have good generalization performance instead of simply memorizing the training dataset. If we only focus on decreasing training loss, we could still have high validation loss, which could yeield bad generalization. When loss stop decreasing, what we need to do next really depends. If training loss converges but validation loss starts to go up, we need to stop training. This can be done through early stopping. We can also further finetune our hyperparameters, hoping to get better convergence. In addition, using more advanced learning rate schedule like cosine annealing schedule could also help.

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.

Answer: For the `BaseConv` model, I just have four Conv-BN-ReLU layers where the spatial dimension of the feature map is kept the same since a segmentation mask requires a label for each pixel. For the number of channels, they are initially increasing, from 3 to 64. At last, I used a 1×1 convolution to reduce the number of channels to the number of classes so that we can get a logit for each class at each pixel. For the convolution layers except the 1×1 convolution, I used a common kernel size of 3 because it achieves a balance between computational efficiency (less parameters & specially optimized implementation in CUDA) and the size of its receptive field (local v.s. global features). The choice of stride and padding depends on my requirement of the spatial dimension of the feature map.

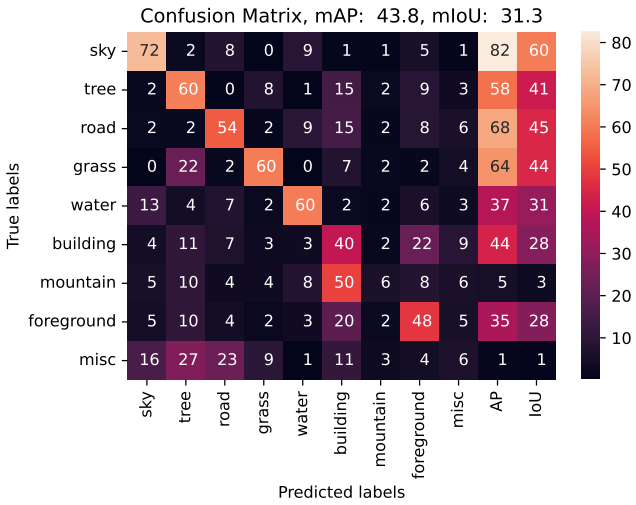


Figure 4: BaseConv Validation Result

Below is the training loss curve of BaseConv. We can observe some degrees of overfitting.

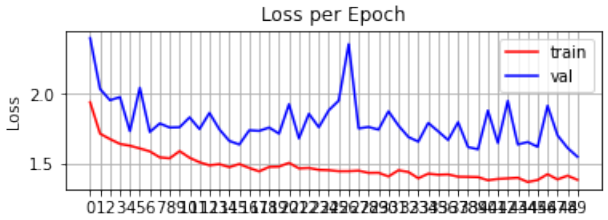


Figure 3: Loss Curve of BaseConv

```
BaseConv(
  (conv): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): Conv2d(64, 9, kernel_size=(1, 1), stride=(1, 1))
  )
)
```

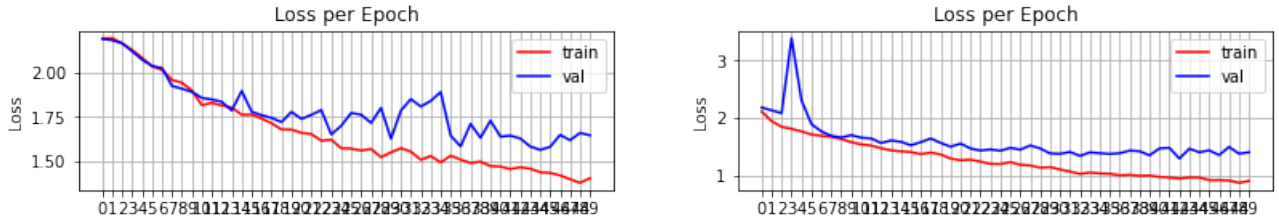
For UNet, I followed the standard UNet architecture but reduced the size of the largest latent vector from 1024 to 512. In the encoder, I used double convolution layers followed by max pooling to reduce the spatial dimension by half and double the depth of the feature map at each layer. In the decoder, I used transposed convolution followed by double convolution layers to double the spatial dimensional and

reduce the depth of the feature map by half at each step, eventually restoring the spatial dimension of the image and obtaining logits for each pixel. I also added long skip connections between the encoder and the decoder where feature maps with the same spatial dimension are concatenated and passed to the downstream double convolutions in the decoder, allowing for smoother optimization landscape and the potential recovery of fine-grained details. In all convolution layer, I used 3×3 kernels except for the last 1×1 convolution layer. The choice of stride and padding depends on my requirement of the spatial dimension of the feature map.

Answer for Notebook Questions: Please check the descriptions about model architecture in the text above. The full architecture is attached after the results. After finalizing the architecture I trained both models for 50 epochs. The number of epochs is determined by observing the convergence of loss curve: we need a convergent training loss while keep an eye on validation loss so that the overfitting is not too severe. I used a batch size of 64 because it balances GPU memory usage and the effect of stochastic gradient descent: on one hand, full batch gradient descent has little or no stochasticity and is expensive; on the other hand, a balanced batch size is cheaper and enables stochastic gradient descent. I chose a learning rate of 0.001 because the batch size is relatively small and it balances between convergence speed and training stability. For both BaseNet and UNet, I used AdamW optimizer. It extends Adam by decoupling weight decay and ℓ_2 regularization, which could potentially improve generalization compared with Adam since Adam was shown to have some failure case in terms of generalization despite good convergence speed.

Batch Normalization helps stabilize and speed up the convergence of very deep neural networks by reducing the internal covariate shift of the input.

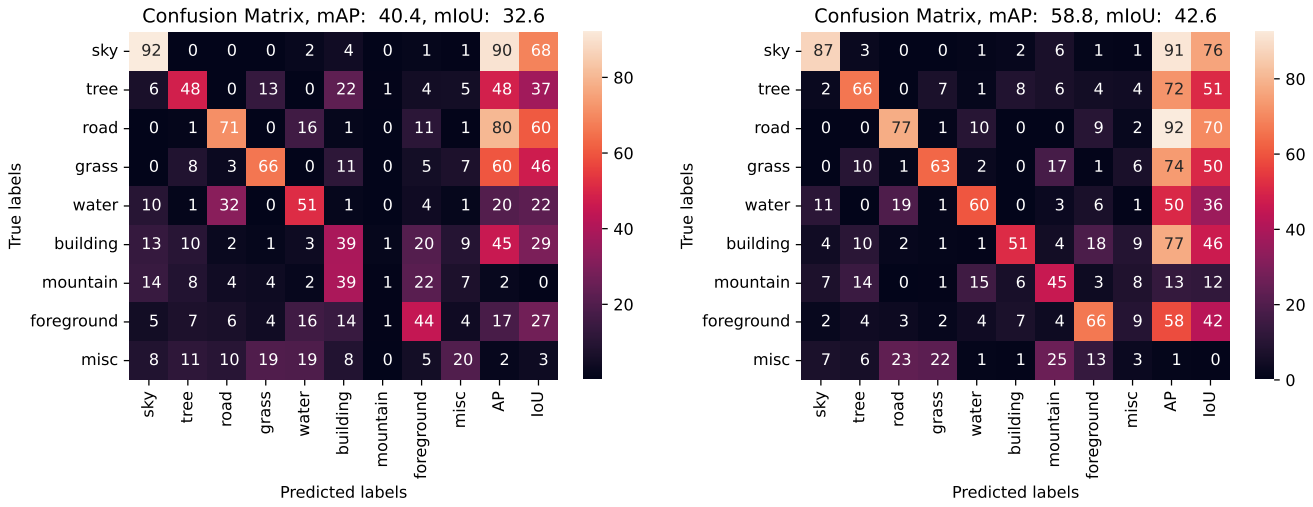
Below are the training loss curves and validation results of UNet with or without Batch Normalization. In both cases, we can observe some degrees of overfitting. Adding batch normalization indeed makes the convergence better and has smoother training trajectory.



(a) Without Batch Normalization

(b) With Batch Normalization

Figure 5: Loss Curve of UNet



(a) Without Batch Normalization

(b) With Batch Normalization

Figure 6: UNet Validation Result

I referred to the following external resources for UNet implementation:
`https://github.com/usuyama/pytorch-unet`
`https://github.com/milesial/Pytorch-UNet/`

Below is a detailed description of the full UNet model.

```
UNet(
  (conv): DoubleConv(
    (double_conv): Sequential(
      (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (down1): Down(
    (maxpool_conv): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (1): DoubleConv(
        (double_conv): Sequential(
          (0): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (5): ReLU(inplace=True)
        )
      )
    )
```

```
)
(down2): Down(
  (maxpool_conv): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (1): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
)
(down3): Down(
  (maxpool_conv): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (1): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
)
(down4): Down(
  (maxpool_conv): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (1): DoubleConv(
      (double_conv): Sequential(
        (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): ReLU(inplace=True)
      )
    )
  )
)
(up1): Up(
  (up): ConvTranspose2d(256, 128, kernel_size=(2, 2), stride=(2, 2))
```

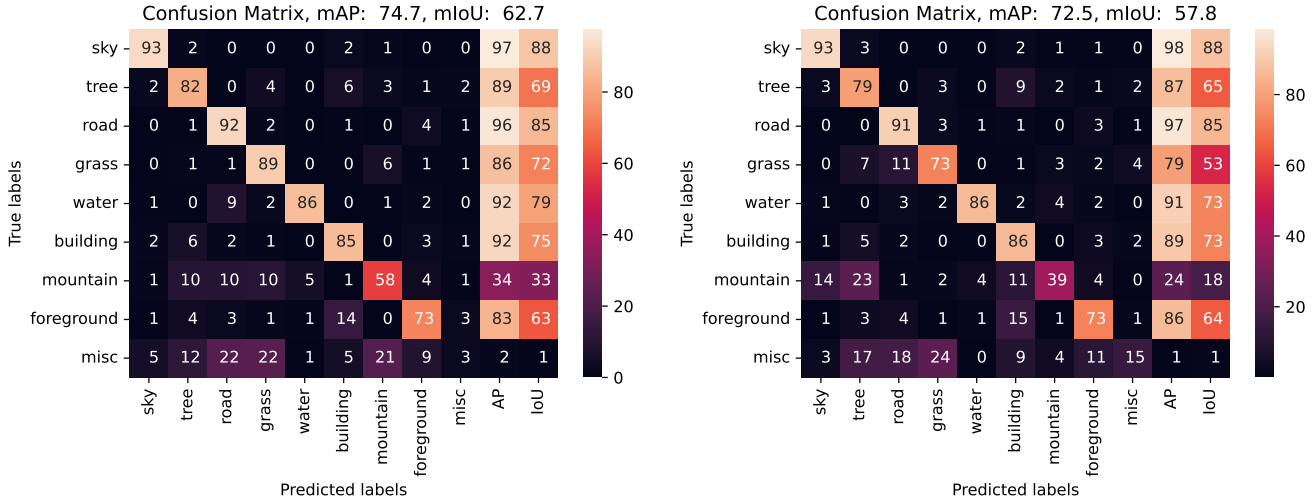
```
  (conv): DoubleConv(
    (double_conv): Sequential(
      (0): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
)
(up2): Up(
  (up): ConvTranspose2d(128, 64, kernel_size=(2, 2), stride=(2, 2))
  (conv): DoubleConv(
    (double_conv): Sequential(
      (0): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
)
(up3): Up(
  (up): ConvTranspose2d(64, 32, kernel_size=(2, 2), stride=(2, 2))
  (conv): DoubleConv(
    (double_conv): Sequential(
      (0): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
)
(up4): Up(
  (up): ConvTranspose2d(32, 16, kernel_size=(2, 2), stride=(2, 2))
  (conv): DoubleConv(
    (double_conv): Sequential(
      (0): Conv2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
```

```
    )
  )
  (clf): Conv1x1(
    (conv): Conv2d(16, 9, kernel_size=(1, 1), stride=(1, 1))
  )
)
```

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.

Answer: To implement a UNet model with pretrained backbone, I designed the decoder architecture where spatial upsampling is consistent with the downsamping pattern in the encoder backbone. In the encoder backbone, the spatial dimension of the feature map is reduced to $\left(\frac{H}{32}, \frac{W}{32}\right)$ and the depth of the feature map is increased to 512 after five ResNet Basic Block. Therefore, in the decoder, I used transposed convolution followed by double convolutions to upscale the spatial dimension and downscale the depth of the feature map such that they can match their counterparts in the encoder. I also added long skip connections between the encoder block and the corresponding and decoder block where feature maps with the same spatial dimension are concatenated and passed to downstream double convolution layers.

Answer for Notebook Questions: Please check how I used the pretrained model from the description above. For the feature I extracted, I threw out the classification head in the pretrained ResNet and only get the latent feature vector with 512 channel dimensions from the backbone. I also extracted intermediate feature maps with smaller depth to concatenate with decoder features in order to implement long skip connections. During the finetuning step, I froze the pretrained parameters in the encoder backbone and update the parameters of decoder for 15 - 30 epochs. The number of epochs is determined by observing the convergence of loss curve: we need a convergent training loss while keep an eye on validation loss so that the overfitting is not too severe. I used a batch size of 64 and a learning rate of 0.001 since the number of parameters to tune is smaller than training from scratch. I used AdamW optimizer following the reasoning from the previous questions.



(a) Validation Result

(b) Test Result

Figure 7: UNet with ResNet Backbone Validation Result

Below is the training loss curve of UNet with ResNet backbone. We can observe some degrees of overfitting.

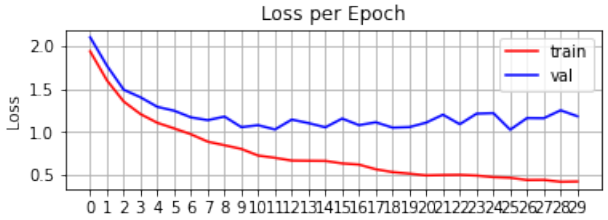


Figure 8: Loss Curve of UNet with ResNet Backbone

Please see the detailed description of the architecture below.

```
ResnetBasedModel(
  (conv): DoubleConv(
    (double_conv): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (layer0): Sequential(
    (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (layer1): Sequential(
    (0): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (1): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
```

```
)
(layer2): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(layer3): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(layer4): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(up4): Up(
  (up): ConvTranspose2d(512, 256, kernel_size=(2, 2), stride=(2, 2))
  (conv): DoubleConv(
    (double_conv): Sequential(
      (0): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
)
(up3): Up(
  (up): ConvTranspose2d(256, 128, kernel_size=(2, 2), stride=(2, 2))
  (conv): DoubleConv(
    (double_conv): Sequential(
      (0): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
)
(up2): Up(
  (up): ConvTranspose2d(128, 64, kernel_size=(2, 2), stride=(2, 2))
  (conv): DoubleConv(
    (double_conv): Sequential(
      (0): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
        (5): ReLU(inplace=True)
      )
    )
  )
  (up1): ConvTranspose2d(64, 64, kernel_size=(2, 2), stride=(2, 2))
  (up0): ConvTranspose2d(128, 64, kernel_size=(2, 2), stride=(2, 2))
  (clf): Sequential(
    (0): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv1x1(
      (conv): Conv2d(64, 9, kernel_size=(1, 1), stride=(1, 1))
    )
  )
)
```

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).