CSC413 Programming Assignment 2 February 28, 2021

Part A: Pooling and Upsampling

1. The code for this question is shown below.

```
class PoolUpsampleNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        padding = kernel // 2

        ############### YOUR CODE GOES HERE ###############
        # construct cnn
        group_one = nn.Sequential(
            nn.Conv2d(in_channels=num_in_channels, out_channels=num_filters,
    kernel_size=kernel, padding=padding),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU()
        )
        group_two = nn.Sequential(
            nn.Conv2d(in_channels=num_filters, out_channels=2*num_filters,
    kernel_size=kernel, padding=padding),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_features=2*num_filters),
            nn.ReLU()
        )
        group_three = nn.Sequential(
            nn.Conv2d(in_channels=2*num_filters, out_channels=num_filters,
    kernel_size=kernel, padding=padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU()
        )
        group_four = nn.Sequential(
            nn.Conv2d(in_channels=num_filters, out_channels=num_colours,
    kernel_size=kernel, padding=padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_features=num_colours),
            nn.ReLU()
        )
        group_five = nn.Sequential(
            nn.Conv2d(in_channels=num_colours, out_channels=num_colours,
    kernel_size=kernel, padding=padding)
        )
        self.modules_list = nn.ModuleList([group_one, group_two, group_three,
    group_four, group_five])

        ###################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        for m in self.modules_list:
          x = m(x)
        return x
        ###################################################

```

2. After 25 epochs using a batch size of 100 and a kernel size of 3, the best validation accuracy is 41.0%. Generally, these results do not look good although it uses different colors to color different objects. For example, according to Figure 1, this model performs badly for the background of the picture like the sky and the ground as it always presents gray sky and green ground.

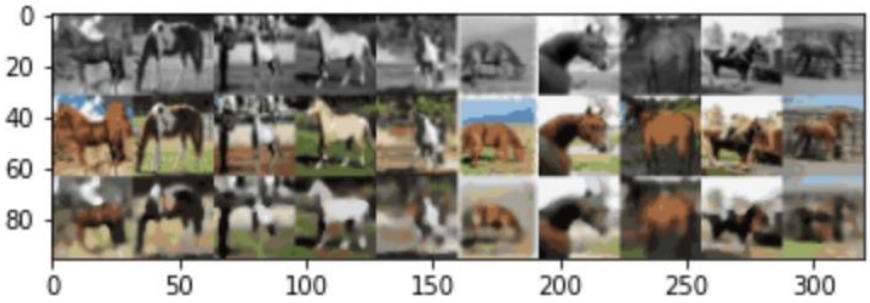

Figure 1: The trained result for some images.

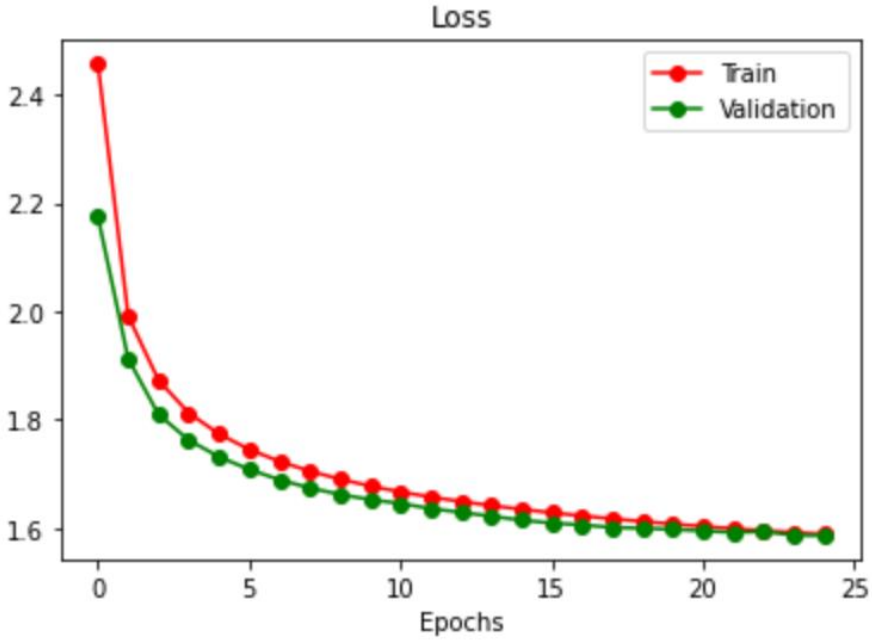

Figure 2: The training and validation loss.

3. For the calculation of the convolution layer, I follow the instruction from the lecture four slide. The number of weights of the max pooling layer, ReLU layer, and up sampling layer are zero since there is no learning parameter. For this model, since the kernel size of the max pooling layer is $2 * 2$, the number of connections is basically the product of the output size and the kernel. Similarly, for the up sampling layer, the number of connections would be its output size because it just scales the tensor by 2. For the batch normalization layer, since the mean and variance of the normalization depends on the feature number, the number of connections is the product of the output size and the size of the height and width.
To make the it clear, let's write it in a table.

   Thus, the total number of weights is

$$9NIC * NF + NF + 2NF + 18NF^2 + 2NF + 4NF + 18NF^2 + NF + 2NF + 9NF * NC + NC + 2NC + 9NC^2$$
$$=36NF^2 + 9NC^2 + 9NF * NC + 9NIC * NF + 12NF + 4NC$$

   The number of outputs is $32 * 32 * NC = 1024NC$.

Layer	Input	Output	# weights	# outputs	# connections
Image	-	$32*32*1$	-	-	-
Conv2d	$32*32*NIC$	$32*32*NF$	$3^2*NIC*NF+NF$	$32*32*NF$	$32*32*3^2*NF*NIC$
MaxPool2d	$32*32*NF$	$16*16*NF$	0	$16*16*NF$	$16*16*2^2*NF$
BatchNorm2d	$16*16*NF$	$16*16*NF$	$2*NF$	$16*16*NF$	$16*16*NF*16^2$
ReLU	$16*16*NF$	$16*16*NF$	0	$16*16*NF$	-
Conv2d	$16*16*NF$	$16*16*2NF$	$3^2*NF*2NF+2NF$	$16*16*2NF$	$16*16*3^2*NF*2NF$
MaxPool2d	$16*16*2NF$	$8*8*2NF$	0	$8*8*2NF$	$8*8*2^2*2NF$
BatchNorm2d	$8*8*2NF$	$8*8*2NF$	$4*NF$	$8*8*2NF$	$8*8*2NF*8^2$
ReLU	$8*8*2NF$	$8*8*2NF$	0	$8*8*2NF$	-
Conv2d	$8*8*2NF$	$8*8*NF$	$3^2*NF*2NF+NF$	$8*8*NF$	$8*8*3^2*NF*2NF$
Upsample	$8*8*NF$	$16*16*NF$	0	$16*16*NF$	$8*8*NF*2^2$
BatchNorm2d	$16*16*NF$	$16*16*NF$	$2*NF$	$16*16*NF$	$16*16*NF*16^2$
ReLU	$16*16*NF$	$16*16*NF$	0	$16*16*NF$	-
Conv2d	$16*16*NF$	$16*16*NC$	$3^2*NF*NC+NC$	$16*16*NC$	$16*16*3^2*NF*NC$
Upsample	$16*16*NC$	$32*32*NC$	0	$32*32*NC$	$16*16*NF*2^2$
BatchNorm2d	$32*32*NC$	$32*32*NC$	$2*NC$	$32*32*NC$	$32*32*NC*32^2$
ReLU	$32*32*NC$	$32*32*NC$	0	$32*32*NC$	-
Conv2d	$32*32*NC$	$32*32*NC$	$3^2*NC*NC+NC$	$32*32*NC$	$32*32*3^2*NC*NC$

The total number of connections is

$$
\begin{aligned}
&9*1024NF*NIC+4*256NF+256*256NF+18*256NF^2+4*128NF+2*64*64NF\\
&+18*64NF^2+4*64NF+256*256NF+9*256NF*NC+4*256NF+1024*1024NC+9*1024NC^2\\
&=9216NF*NIC+9216NC^2+5760NF^2+2304NF*NC+142080NF+1048576NC
\end{aligned}
$$

Similarly, for the doubled input dimension, we can get the following table.

Layer	Input	Output	# weights	# outputs	# connections
Image	-	$64*64*1$	-	-	-
Conv2d	$64*64*NIC$	$64*64*NF$	$3^2*NIC*NF+NF$	$64*64*NF$	$64*64*3^2*NF*NIC$
MaxPool2d	$64*64*NF$	$32*32*NF$	0	$32*32*NF$	$32*32*2^2*NF$
BatchNorm2d	$32*32*NF$	$32*32*NF$	$2*NF$	$32*32*NF$	$32*32*NF*32^2$
ReLU	$32*32*NF$	$32*32*NF$	0	$32*32*NF$	-
Conv2d	$32*32*NF$	$32*32*2NF$	$3^2*NF*2NF+2NF$	$32*32*2NF$	$32*32*3^2*NF*2NF$
MaxPool2d	$32*32*2NF$	$16*16*2NF$	0	$16*16*2NF$	$16*16*2^2*2NF$
BatchNorm2d	$16*16*2NF$	$16*16*2NF$	$4*NF$	$16*16*2NF$	$16*16*2NF*16^2$
ReLU	$16*16*2NF$	$16*16*2NF$	0	$16*16*2NF$	-
Conv2d	$16*16*2NF$	$16*16*NF$	$3^2*NF*2NF+NF$	$16*16*NF$	$16*16*3^2*NF*2NF$
Upsample	$16*16*NF$	$32*32*NF$	0	$32*32*NF$	$16*16*NF*2^2$
BatchNorm2d	$32*32*NF$	$32*32*NF$	$2*NF$	$32*32*NF$	$32*32*NF*32^2$
ReLU	$32*32*NF$	$32*32*NF$	0	$32*32*NF$	-
Conv2d	$32*32*NF$	$32*32*NC$	$3^2*NF*NC+NC$	$32*32*NC$	$32*32*3^2*NF*NC$
Upsample	$32*32*NC$	$64*64*NC$	0	$64*64*NC$	$32*32*NF*2^2$
BatchNorm2d	$64*64*NC$	$64*64*NC$	$2*NC$	$64*64*NC$	$64*64*NC*64^2$
ReLU	$64*64*NC$	$64*64*NC$	0	$64*64*NC$	-
Conv2d	$64*64*NC$	$64*64*NC$	$3^2*NC*NC+NC$	$64*64*NC$	$64*64*3^2*NC*NC$

Thus, the total number of weights is

$$
\begin{aligned}
&9NIC*NF+NF+2NF+18NF^2+2NF+4NF+18NF^2+NF+2NF+9NF*NC+NC+2NC+9NC^2\\
&=36NF^2+9NC^2+9NF*NC+9NIC*NF+12NF+4NC
\end{aligned}
$$

The number of outputs is $64 * 64 * NC = 4096NC$.
The total number of connections is

$$9 * 4096NF * NIC + 4 * 1024NF + 1024 * 1024NF + 18 * 1024NF^2 + 4 * 512NF + 2 * 256 * 256NF$$
$$+18 * 256NF^2 + 4 * 256NF + 1024 * 1024NF + 9 * 1024NF * NC + 4 * 1024NF + 4096 * 4096NC + 9 * 4096NC^2$$
$$=36864NF * NIC + 36864NC^2 + 23040NF^2 + 9216NF * NC + 142080NF + 2239488NC$$

Part B: Strided and Transposed Convolutions

1. The code for this question is shown below.

```
class ConvTransposeNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        stride = 2
        padding = kernel // 2
        output_padding = 1

        ################ YOUR CODE GOES HERE ################
        group_one = nn.Sequential(
            nn.Conv2d(in_channels=num_in_channels, out_channels=num_filters,
    kernel_size=kernel, stride=2, padding=1),
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU()
        )
        group_two = nn.Sequential(
            nn.Conv2d(in_channels=num_filters, out_channels=2*num_filters,
    kernel_size=kernel, stride=2, padding=1),
            nn.BatchNorm2d(num_features=2*num_filters),
            nn.ReLU()
        )
        group_three = nn.Sequential(
            nn.ConvTranspose2d(in_channels=2*num_filters, out_channels=num_filters
    , kernel_size=kernel, stride=2, padding=1, output_padding=1),
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU()
        )
        group_four = nn.Sequential(
            nn.ConvTranspose2d(in_channels=num_filters, out_channels=num_colours,
    kernel_size=kernel, stride=2, padding=1, output_padding=1),
            nn.BatchNorm2d(num_features=num_colours),
            nn.ReLU()
        )
        group_five = nn.Sequential(
            nn.Conv2d(in_channels=num_colours, out_channels=num_colours,
    kernel_size=kernel, padding=padding)
        )
        self.modules_list = nn.ModuleList([group_one, group_two, group_three,
    group_four, group_five])
        ########################################################

    def forward(self, x):
        ################ YOUR CODE GOES HERE ################
        for m in self.modules_list:
          x = m(x)
        return x
        ########################################################

```

2. After 25 epochs using a batch size of 100 and a kernel size of 3, the best validation accuracy is 55.7%.
3. The result is better than Part A. According to Figure 3, this model performs better for the background of the picture like the sky, but it still always presents green ground. According to

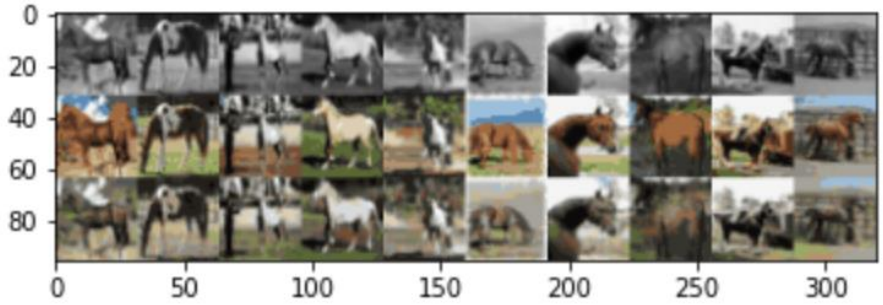

Figure 3: The trained result for some images.

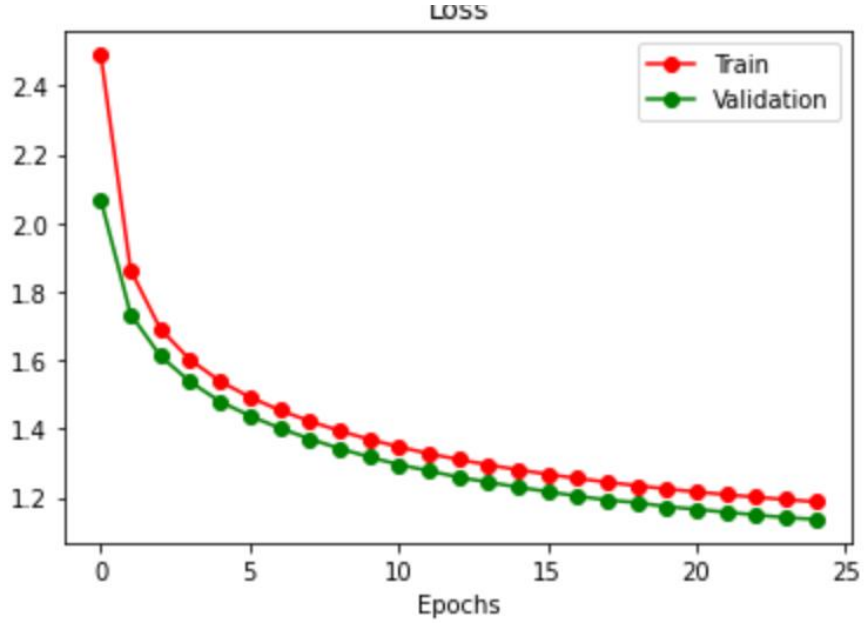

Figure 4: The training and validation loss.

Figure 2 and 4, we can see that the validation loss is always smaller than the training loss for the $ConvTransposeNet$ model. Compared to the $PoolUpsampleNet$ model, the validation loss in the $ConvTransposeNet$ model is lower.

4. Let's consider the padding parameter for the first two convolution layers first. According to the previous setting, we know $padding = 1$ and $stride = 2$ for the case of $kernel\_size = 3$. For the dilation parameter, we use the default setting which is 1. Thus, define a be the input size, b be the output size, p be the padding size, and k be the kernel size. We can get

$$\begin{aligned} b &= \frac{a + 2p - 1 * (k-1) - 1}{2} + 1 \\ &= \frac{a + 2 * 1 - (3-1) - 1}{2} + 1 \\ &= \frac{a+1}{2} \end{aligned}$$

To maintain the output size, it must follow

$$\begin{aligned} 2p - (k-1) + 1 &= 1 \\ \implies 2p - k &= -1 \\ \implies p &= \frac{k-1}{2} \end{aligned}$$

Thus, when $k = 4$, $p = 1.5$. However, 1.5 is invalid for the number of padding. Thus, we should either set $p = 1$ or $p = 2$. When $k = 5$, $p = 2$.
For the padding parameters of the $ConvTranspose2d$ layer, we know $padding = 1$, $stride = 2$, and $output\_padding = 1$ for the case of $kernel\_size = 3$. Similarly, for the dilation parameter, we use the default setting which is 1. Define o be the size of output padding. We can get

$$\begin{aligned} b &= s(a-1) - 2p + 1 * (k-1) + o + 1 \\ &= 2a - 2 - 2 + 2 + 1 + 1 \\ &= 2a \end{aligned}$$

To maintain the output size, it must follow

$$\begin{aligned} -2p + k + o - 2 &= 0 \\ \implies 2p - o &= k - 2 \end{aligned}$$

Thus, there are many solutions. When $k = 4$, $2p - o = 2$. For example, we can set $padding = 1$ and $output\_padding = 0$. When $k = 5$, $2p - o = 3$. Setting $padding = 2$ and $output\_padding = 1$ works.

5. I set the batch size be $32, 64, 128, 256$, and 512 with 25 epochs. The following table shows the corresponding best validation loss.

Batch size	Best validation loss
32	57.4%
64	55.2%
128	53.1%
256	50.6%
512	45.9%

According to the result, we can see that the model performs worse as we increase the batch size. When the batch size is 32, it achieved the highest accuracy as well as the best final image output comparing with others. As the batch size becomes larger, the difference between the training loss and the validation loss becomes smaller. Thus, we can conclude that the model prefers small value of the batch size.

Part C: Skip Connections

1. The code for this question is shown below.

```
class UNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        stride = 2
        padding = kernel // 2
        output_padding = 1

        ################ YOUR CODE GOES HERE ################
        self.group_one = nn.Sequential(
            nn.Conv2d(in_channels=num_in_channels, out_channels=num_filters,
    kernel_size=kernel, stride=stride, padding=padding),
```

```
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU()
        )
        self.group_two = nn.Sequential(
            nn.Conv2d(in_channels=num_filters, out_channels=2*num_filters,
    kernel_size=kernel, stride=stride, padding=padding),
            nn.BatchNorm2d(num_features=2*num_filters),
            nn.ReLU()
        )
        self.group_three = nn.Sequential(
            nn.ConvTranspose2d(in_channels=2*num_filters, out_channels=num_filters
    , kernel_size=kernel, stride=stride, padding=padding, output_padding=
    output_padding),
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU()
        )
        self.group_four = nn.Sequential(
            nn.ConvTranspose2d(in_channels=2*num_filters, out_channels=num_colours
    , kernel_size=kernel, stride=2, padding=padding, output_padding=output_padding
    ),
            nn.BatchNorm2d(num_features=num_colours),
            nn.ReLU()
        )
        self.group_five = nn.Sequential(
            nn.Conv2d(in_channels=num_in_channels+num_colours, out_channels=
    num_colours, kernel_size=kernel, padding=padding)
        )
        ########################################################

    def forward(self, x):
        ################ YOUR CODE GOES HERE ################
        group_one_out = self.group_one(x)
        group_two_out = self.group_two(group_one_out)
        group_three_out = self.group_three(group_two_out)
        group_four_in = torch.cat((group_one_out, group_three_out), 1)
        group_four_out = self.group_four(group_four_in)
        group_five_in = torch.cat((x, group_four_out), 1)
        return self.group_five(group_five_in)
        ########################################################

```

2. After 25 epochs using a batch size of 100 and a kernel size of 3, the accuracy is 58.2%.

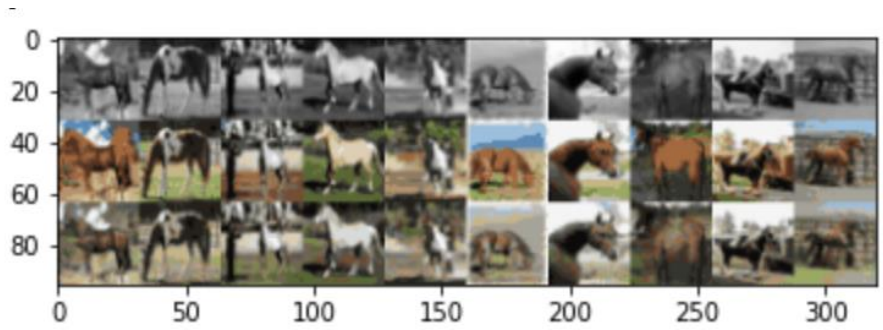

Figure 5: The trained result for some images.

3. The result is better than both Part A and Part B. Thus, we can conclude that the skip connections improve the accuracy and the output qualitatively. According to Figure 4, this model performs better for the background of the picture like the sky, and it performs better about the ground as it uses different colors to paint ground. According to Figure 2, 4 and 6, we can see that the validation loss is always smaller than the training loss for the $UNet$ model. Compared to the $PoolUpsampleNet$ and $ConvTransposeNet$ model, the validation loss in the $UNet$ model is

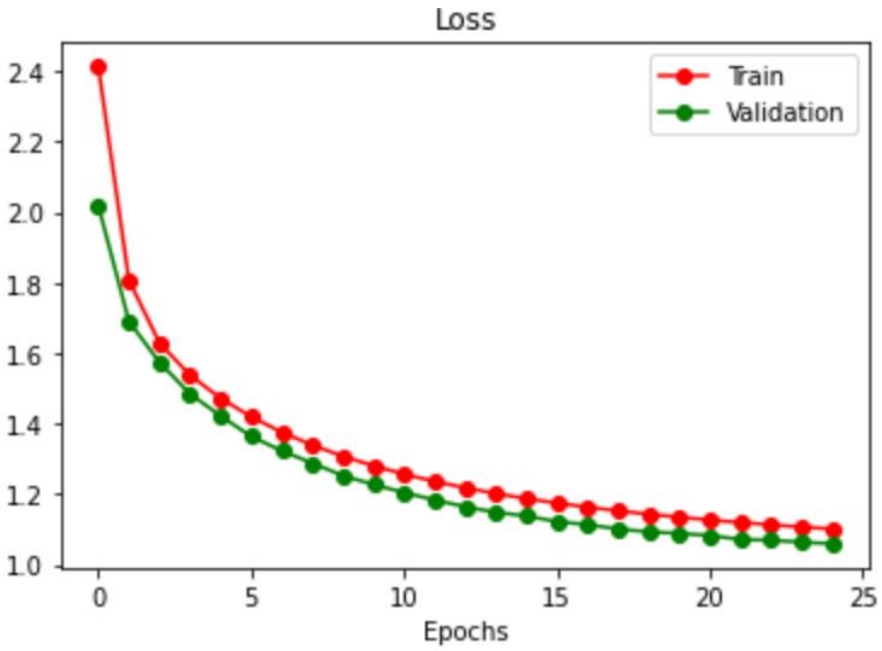

Figure 6: The training and validation loss.

lower. I think one of the reasons is that skip connections require more trainable parameters since we concatenate two layers in a pattern, which can learn more things from the input and perform better. The other reason might be concatenating two layers makes up for the possible missing information from the pooling or sampling process.

Part D.1: Fine-tune Semantic Segmentation Model with Cross Entropy Loss

1. The code for this question is shown below.

```
def train(args, model):

    # Set the maximum number of threads to prevent crash in Teaching Labs
    torch.set_num_threads(5)
    # Numpy random seed
    np.random.seed(args.seed)

    # Save directory
    # Create the outputs folder if not created already
    save_dir = "outputs/" + args.experiment_name
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

    learned_parameters = []
    # We only learn the last layer and freeze all the other weights
    ################## Code goes here ######################
    # Around 3 lines of code
    # Hint:
    # - use a for loop to loop over all model.named_parameters()
    # - append the parameters (both weights and biases) of the last layer (prefix:
      classifier.4) to the learned_parameters list
    for name, param in model.named_parameters():
      if name.startswith("classifier.4"):
        learned_parameters.append(param)
    ########################################################

```

2. The code for this question is shown below.

```
# Truncate the last layer and replace it with the new one.
# To avoid 'CUDA out of memory' error, you might find it useful (sometimes
    required)
#   to set the 'requires_grad'=False for some layers
################## YOUR CODE GOES HERE ######################
# Around 4 lines of code
# Hint:
# - replace the classifier.4 layer with the new Conv2d layer (1 line)
# - no need to consider the aux_classifier module (just treat it as don't care)
# - freeze the gradient of other layers (3 lines)
model.classifier._modules['4'] = nn.Conv2d(256, 2, kernel_size=(1, 1), stride=(1,
    1))
for name, param in model.named_parameters():
  if not name.startswith("classifier.4"):
    param.requires_grad = False
#############################################################

```

According to the output, the best validation mIoU is 0.3373.

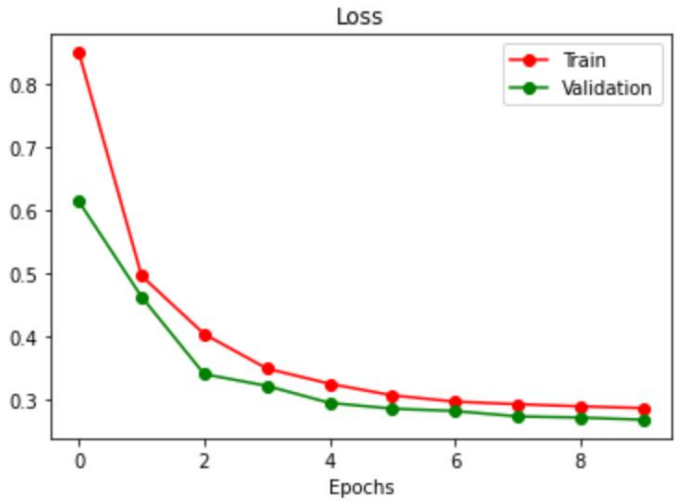

Figure 7: The training and validation loss.

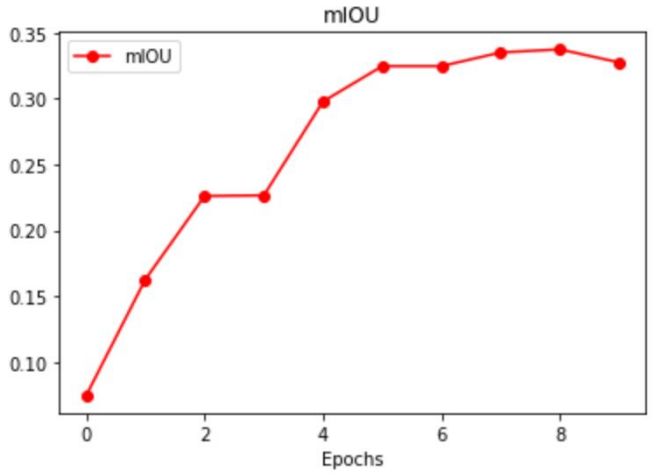

Figure 8: The mIoU.

3. Visualization

Figure 9: The predictions visualization.

Part D.2: Fine-tune Semantic Segmentation Model with IoU Loss

1. The code for this question is shown below.

```
def compute_iou_loss(pred, gt, SMOOTH=1e-6):
    # Compute the IoU between the pred and the gt (ground truth)
    ################## YOUR CODE GOES HERE ########################
    # Around 5 lines of code
    # Hint:
    # - apply softmax on pred along the channel dimension (dim=1)
    # - only have to compute IoU between gt and the foreground channel of pred
    # - no need to consider IoU for the background channel of pred
    # - extract foreground from the softmaxed pred (e.g., softmaxed_pred[:, 1, :,
    :])
    # - compute intersection between foreground and gt
    # - compute union between foreground and gt
    # - compute loss using the computed intersection and union
    softmaxed_pred = F.softmax(pred, dim=1)
    new_pred = softmaxed_pred[:, 1, :, :]
    i = torch.mul(new_pred, gt)
    u = new_pred + gt - i
    loss = 1 - torch.sum(i)/torch.sum(u)
    ###############################################################

    return loss

```

According to the output, the best validation mIoU is 0.3014. It performs slightly worse than the previous model although we use stronger loss function. However, according to Figure 10, we can see that the corresponding training loss and validation loss are about 0.4, which seems not to be converged and could be improved. Thus, if we tune some hyperparameters like the number of epochs, we might get better result.

2. Visualization

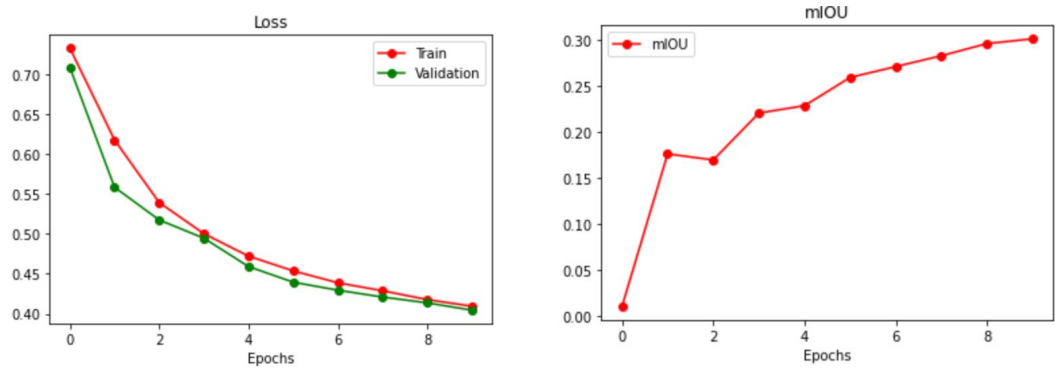

Figure 10: The loss and mIoU.

Figure 11: The predictions visualization.