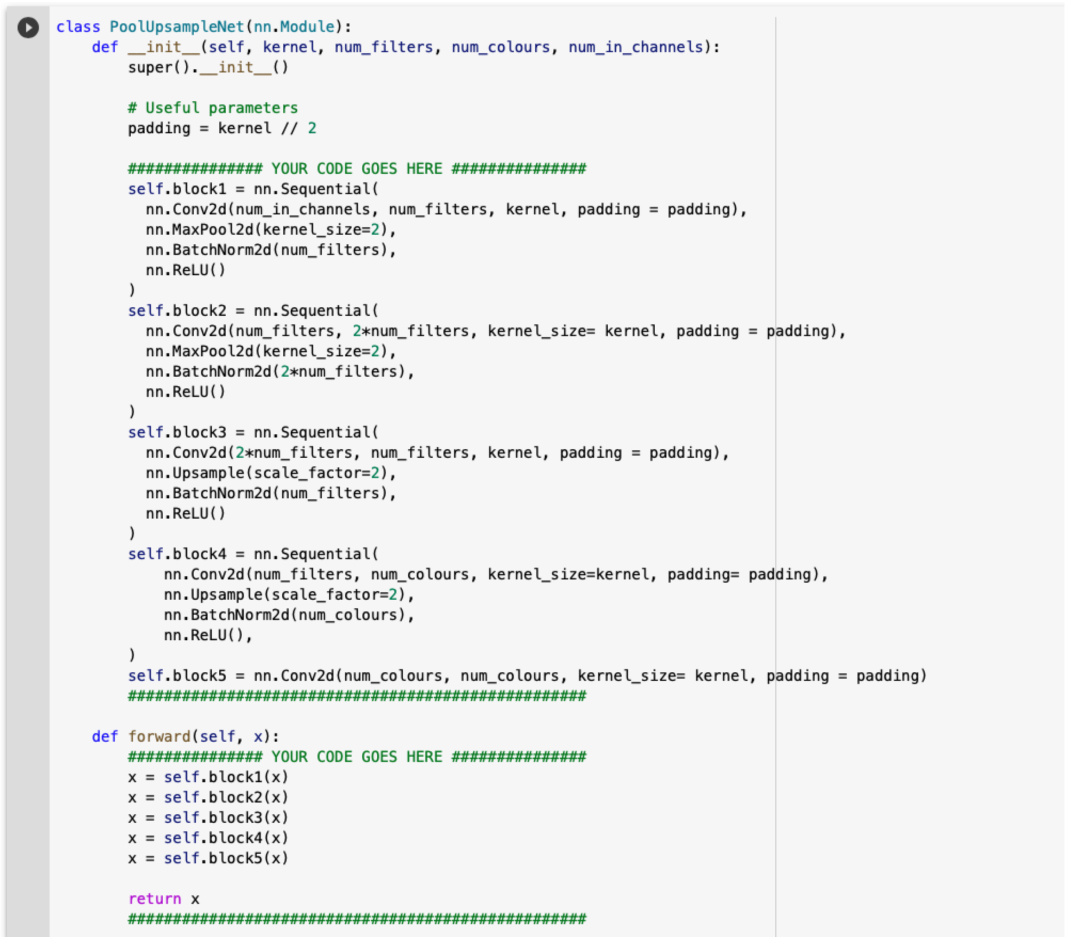
Programming Assignment 2

1 Part A: Pooling and Upsampling

1.1

Look at Fig 1.



```python
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 ###############
        self.block1 = nn.Sequential(
          nn.Conv2d(num_in_channels, num_filters, kernel, padding = padding),
          nn.MaxPool2d(kernel_size=2),
          nn.BatchNorm2d(num_filters),
          nn.ReLU()
        )
        self.block2 = nn.Sequential(
          nn.Conv2d(num_filters, 2*num_filters, kernel_size= kernel, padding = padding),
          nn.MaxPool2d(kernel_size=2),
          nn.BatchNorm2d(2*num_filters),
          nn.ReLU()
        )
        self.block3 = nn.Sequential(
          nn.Conv2d(2*num_filters, num_filters, kernel, padding = padding),
          nn.Upsample(scale_factor=2),
          nn.BatchNorm2d(num_filters),
          nn.ReLU()
        )
        self.block4 = nn.Sequential(
            nn.Conv2d(num_filters, num_colours, kernel_size=kernel, padding= padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_colours),
            nn.ReLU(),
        )
        self.block5 = nn.Conv2d(num_colours, num_colours, kernel_size= kernel, padding = padding)
        ###################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.block4(x)
        x = self.block5(x)

        return x
        ###################################################
```

Figure 1: PoolUpsampleNet implementation

1.2

Run main training loop of PoolUpsampleNet. This will train the CNN for a few epochs using the cross-entropy objective. It will generate some images showing the trained result at the end. Do these results look good to you? Why or why not?

Look at Fig 2 After running the main training loop of PoolUpsampleNet, we can see that the results are not the best. Even though the loss decreases from 2.4453 to 1.5742 after 25 epochs, the validation accuracy of the model is quite low (41%).
Furthermore, comparing the trained image colourization results versus the ground truth colourized

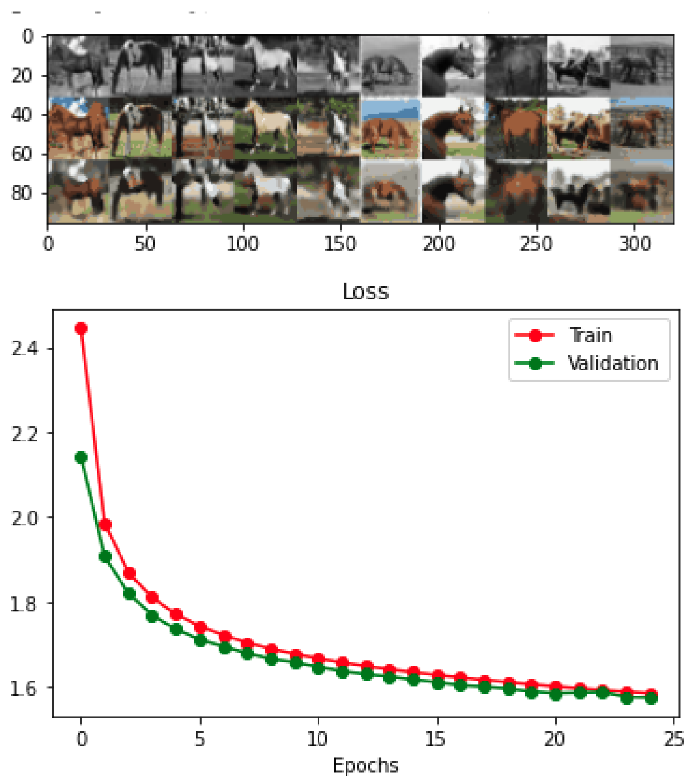


Figure 2: Caption

image, we can see that our model doesn't give the most accurate predictions. We can see that the model performs fairly poorly on predicting diverse/stark colours in an image. A better model can be implemented.

1.3

Compute the number of weights, outputs, and connections in the model, as a function of NIC, NF and NC. Compute these values when each input dimension (width/height) is doubled. Report all 6 values.

When input dimension is 32×32:

Total connections: $2432NF + 5760NF^2 + 2048NC + 2304NFNC + 9216NC^2 + 9216NICNF$
Total Output Units: $2880NF + 3328NC$
Total Weights: $9NICNF + 36NF^2 + 9NCNF + 9NC^2$

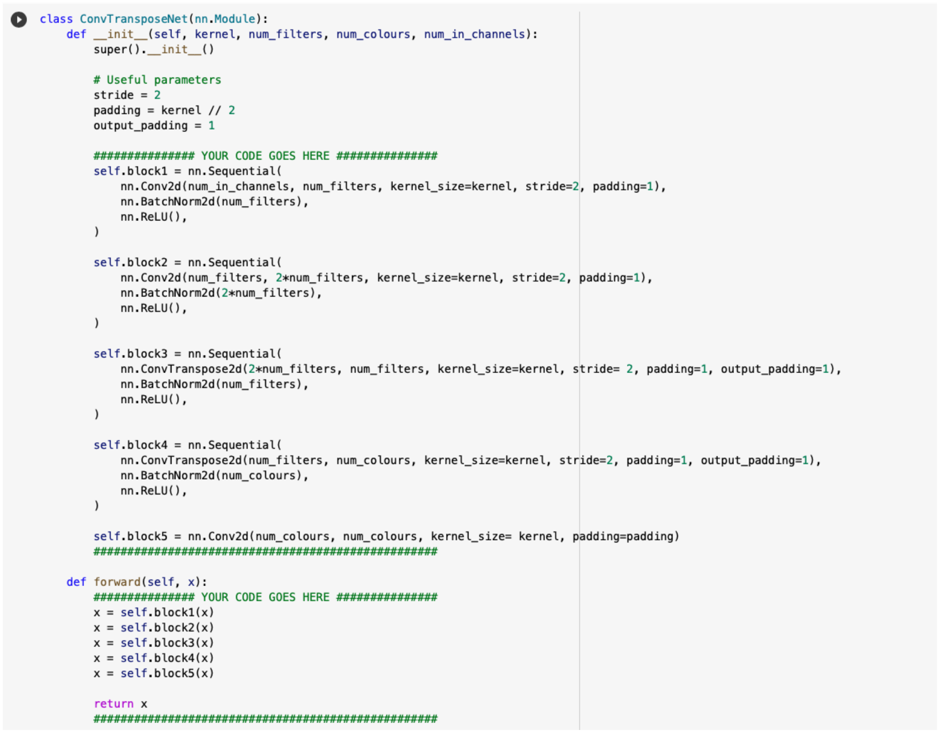
When input dimension is doubled(64×64):

Total connections: $9728NF + 8192NC + 36864NICNF + 23040NF^2 + 9216NFNC + 36864NC^2$
Total Output Units $= 11520NF + 13312NC$
Total Weights: $9NICNF + 36NF^2 + 9NCNF + 9NC^2$

2 Part B: Strided and Transposed Convolutions

2.1

Look at Fig 3



```python
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 ###############
        self.block1 = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, stride=2, padding=1),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(),
        )

        self.block2 = nn.Sequential(
            nn.Conv2d(num_filters, 2*num_filters, kernel_size=kernel, stride=2, padding=1),
            nn.BatchNorm2d(2*num_filters),
            nn.ReLU(),
        )

        self.block3 = nn.Sequential(
            nn.ConvTranspose2d(2*num_filters, num_filters, kernel_size=kernel, stride= 2, padding=1, output_padding=1),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(),
        )

        self.block4 = nn.Sequential(
            nn.ConvTranspose2d(num_filters, num_colours, kernel_size=kernel, stride=2, padding=1, output_padding=1),
            nn.BatchNorm2d(num_colours),
            nn.ReLU(),
        )

        self.block5 = nn.Conv2d(num_colours, num_colours, kernel_size= kernel, padding=padding)
        ###################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.block4(x)
        x = self.block5(x)

        return x
        ###################################################
```

Figure 3: ConvTransposeNet implementation

2.2

Look at Fig 4

2.3

How do the result compare to Part A? Does the ConvTransposeNet model result in lower validation loss than the PoolUpsampleNet? Why may this be the case?

Results in Part B seem better than the results from Part A, i.e., the ConvTransposeNet network performs better than the PoolUpsampleNet network. The trained images for the ConvTransposeNet network look to be closer to the ground truth images as compared to the trained images from the model in Part A.
Yes, ConvTransposeNet results in a lower validation loss than PoolUpsampleNet and a higher validation accuracy (validation accuracy increased by about 15%). This is because, we replace the max pool layers with strided convolution layers, this makes the model do better. Maxpool is used to decrease spatial resolution but using maxpool layers involve some information loss (same goes for Upsample layers). Strided convolution/transposed convolution layers allow us to perform

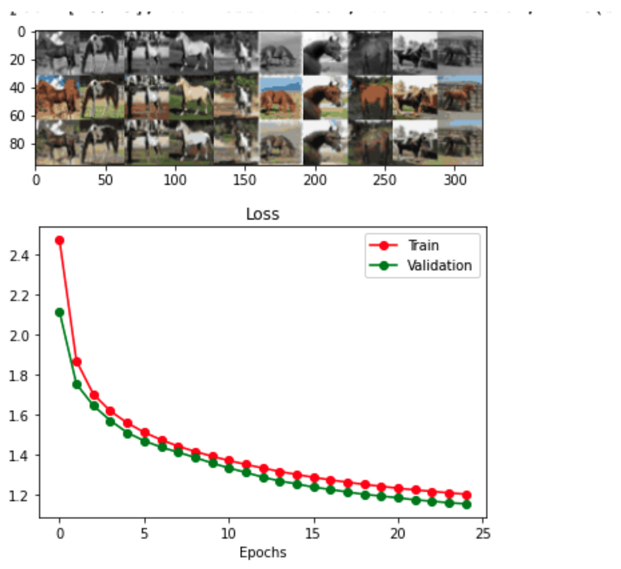


Figure 4:

same task as Maxpool/Upsample layers in a data driven fashion a these strided convolution layers contain learnable parameters as opposed to MaxPool layers which do not contain any learnable parameters (same with Upsample layers).
Reasons mentioned above make the ConvTransposedNet do better than PoolUpsampleNet.

2.4

How would the padding parameter passed to the first two nn.Conv2d layers, and the padding and output padding parameters passed to the nn.ConvTranspose2d layers, need to be modified if we were to use a kernel size of 4 or 5?

To maintain shapes of all tensors shown in Figure 1b, we need to choose padding size for both kinds of layers given the kernel size $k = 4$, $stride = 2$.

For 1st 2 nn.Conv2d layers, if kernel size is 4, then choose padding size = 1 (i.e. set padding = 1) to maintain tensor sizes shown in Figure 1b).

For nn.ConvTranspose2d layers, let *padding* be x; let *output_padding* be x_1. Then if kernel size $k = 4$, we get result that:
$2x = x_1 + 2$
Therefore, if we choose $x = 10$ (padding = 10), then $x_1 = 18$ (output padding = 18), our result follows. Now, if we have kernel size given to be 4 then we setting padding/output padding parameters as mentioned above will allow us to maintain tensor sizes shown in Figure 1b).

2.5

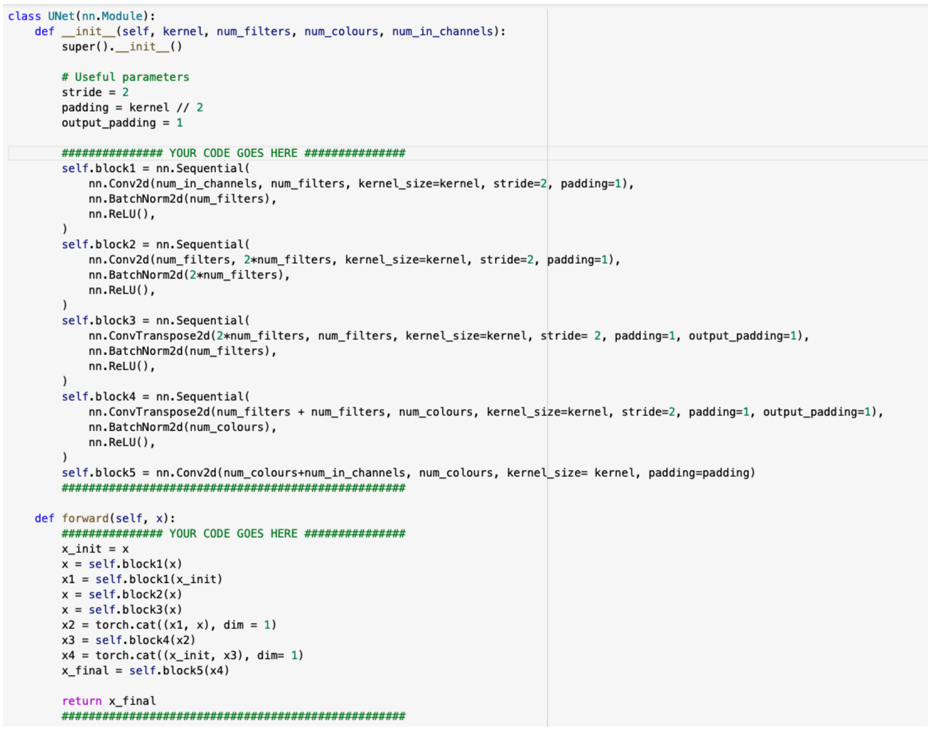
Effect of batch sizes on the training/validation loss, and final image output quality:
As the batch size increases, the validation accuracy goes down. The training and validation loss both increase as the batch size increases which tells us the network performs worse as the batch

size is increased. The trained image quality also becomes poorer and poorer as the batch size is increased.

3 Part C: Skip connections

3.1

Look at Fig 5



```
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.block1 = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, stride=2, padding=1),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(),
        )
        self.block2 = nn.Sequential(
            nn.Conv2d(num_filters, 2*num_filters, kernel_size=kernel, stride=2, padding=1),
            nn.BatchNorm2d(2*num_filters),
            nn.ReLU(),
        )
        self.block3 = nn.Sequential(
            nn.ConvTranspose2d(2*num_filters, num_filters, kernel_size=kernel, stride= 2, padding=1, output_padding=1),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(),
        )
        self.block4 = nn.Sequential(
            nn.ConvTranspose2d(num_filters + num_filters, num_colours, kernel_size=kernel, stride=2, padding=1, output_padding=1),
            nn.BatchNorm2d(num_colours),
            nn.ReLU(),
        )
        self.block5 = nn.Conv2d(num_colours+num_in_channels, num_colours, kernel_size= kernel, padding=padding)
        ###################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        x_init = x
        x = self.block1(x)
        x1 = self.block1(x_init)
        x = self.block2(x)
        x = self.block3(x)
        x2 = torch.cat((x1, x), dim = 1)
        x3 = self.block4(x2)
        x4 = torch.cat((x_init, x3), dim= 1)
        x_final = self.block5(x4)

        return x_final
        ###################################################
```

Figure 5: UNet implementation

3.2

Look at Fig 6

3.3

The result is fairly better than the previous results quantitatively. The validation accuracy achieved with this network is about 58% which is 3% better than the validation accuracy we saw with ConvTransposeNet. Qualitatively, our trained image predictions now seem to be slightly closer to the ground truth than the predictions we saw with earlier networks. Yes, the skip connections did improve the quality of our output. The colour recognition seems more drastic than what we saw in PoolUpsampleNet.

The skip connections may improve CNN performance by:

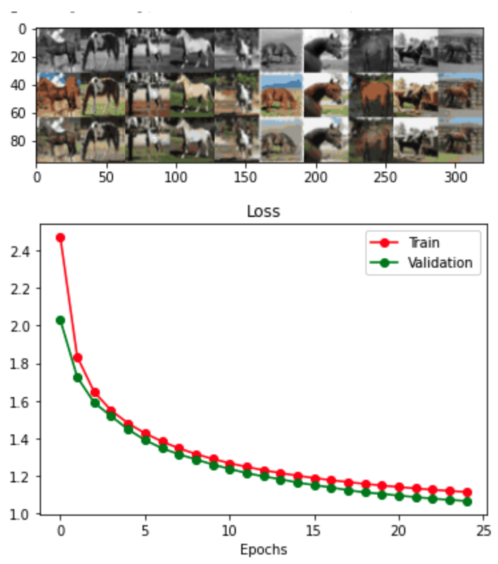


Figure 6:

1.) providing an alternative path for the gradient; in dense networks, sometimes, the early layers are not updated as the gradient becomes very small. Adding skip connections allows direct connections between early and deeper layers in a network allowing for better updating of weight parameters.
2.) allowing for recognition of more diverse and deep features of an input; skipping between layers allows the deeper layers of our model to look at the original input as well as the input to the layer generated by the network and this allows for more information being recognized and processed by the network which further improves performance as the network learns better (and more).

4 Part D.1

4.1

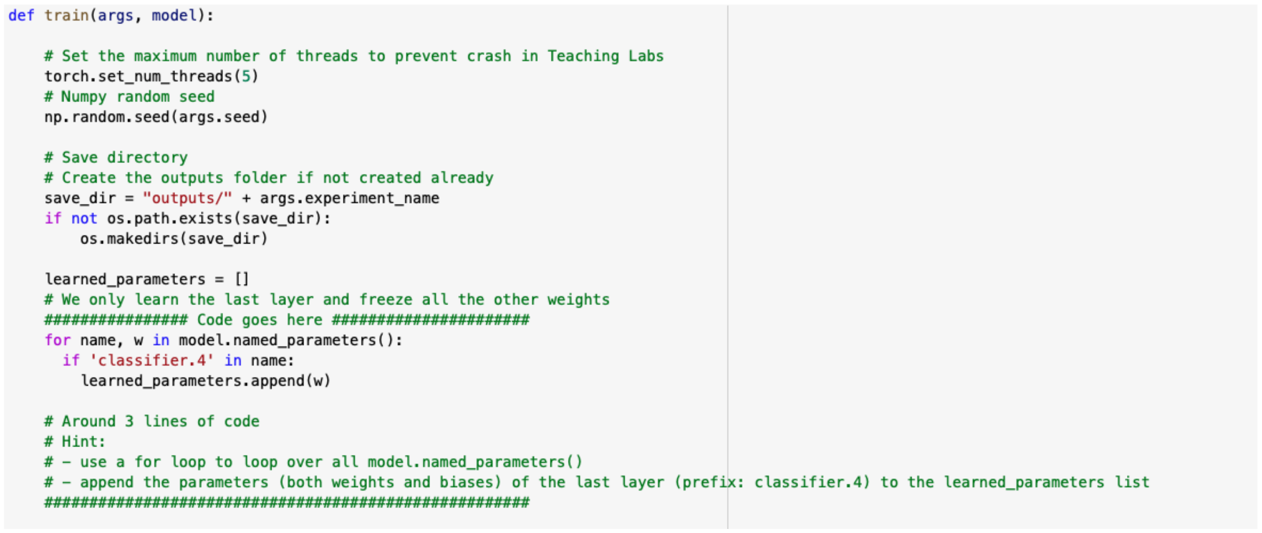
Look at Fig 7

4.2

Look at Fig 8
Best validation mIOU: 0.3500

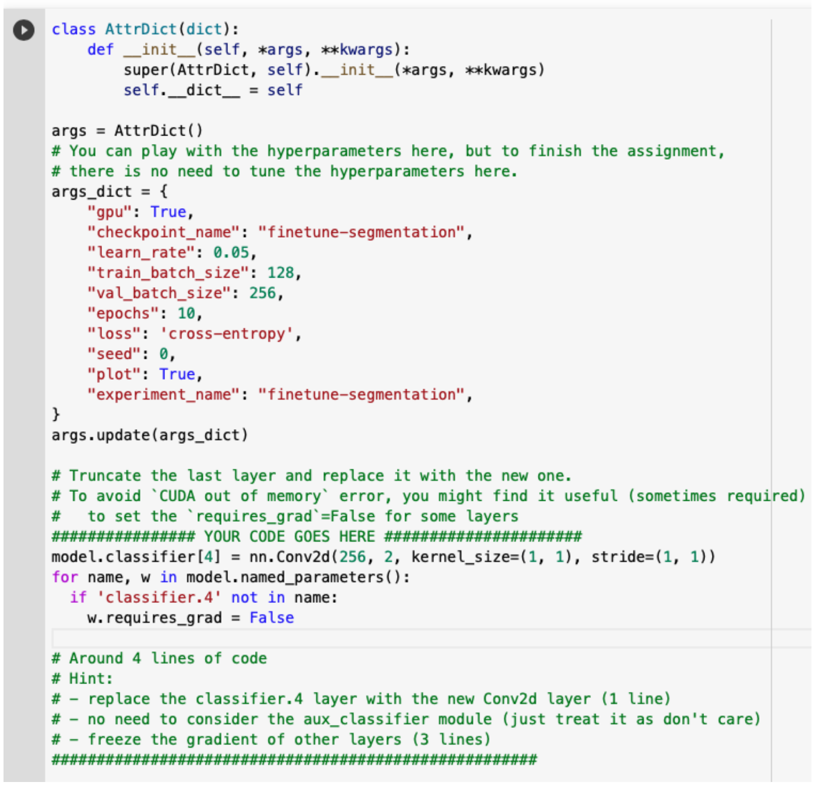
4.3

Look at Fig 9



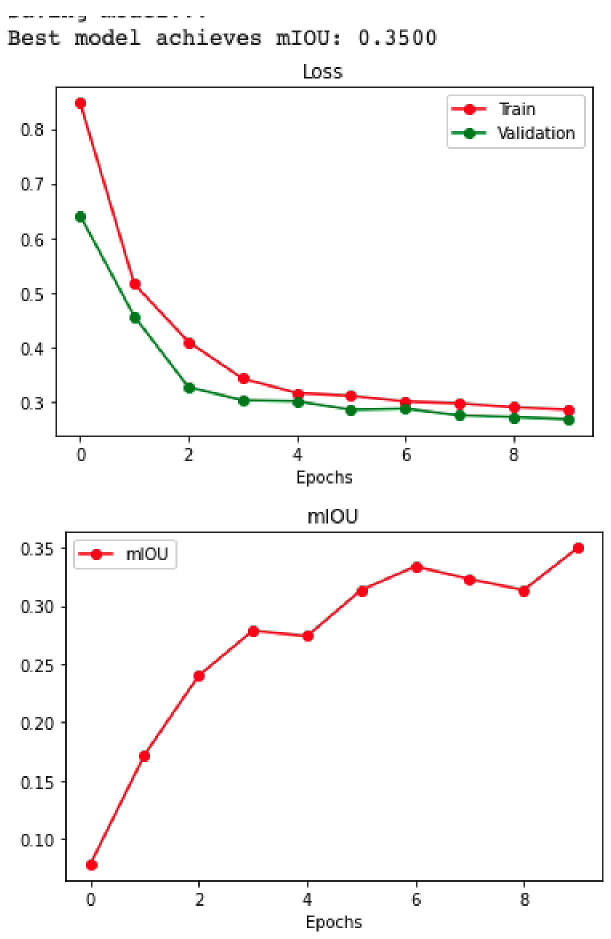
```
def train(args, model):

    # 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 ######################
    for name, w in model.named_parameters():
      if 'classifier.4' in name:
        learned_parameters.append(w)

    # 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
    ######################################################
```

Figure 7:



```
class AttrDict(dict):
    def __init__(self, *args, **kwargs):
        super(AttrDict, self).__init__(*args, **kwargs)
        self.__dict__ = self

args = AttrDict()
# You can play with the hyperparameters here, but to finish the assignment,
# there is no need to tune the hyperparameters here.
args_dict = {
    "gpu": True,
    "checkpoint_name": "finetune-segmentation",
    "learn_rate": 0.05,
    "train_batch_size": 128,
    "val_batch_size": 256,
    "epochs": 10,
    "loss": 'cross-entropy',
    "seed": 0,
    "plot": True,
    "experiment_name": "finetune-segmentation",
}
args.update(args_dict)

# 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 ######################
model.classifier[4] = nn.Conv2d(256, 2, kernel_size=(1, 1), stride=(1, 1))
for name, w in model.named_parameters():
  if 'classifier.4' not in name:
    w.requires_grad = False

# 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)
######################################################
```

Figure 8:



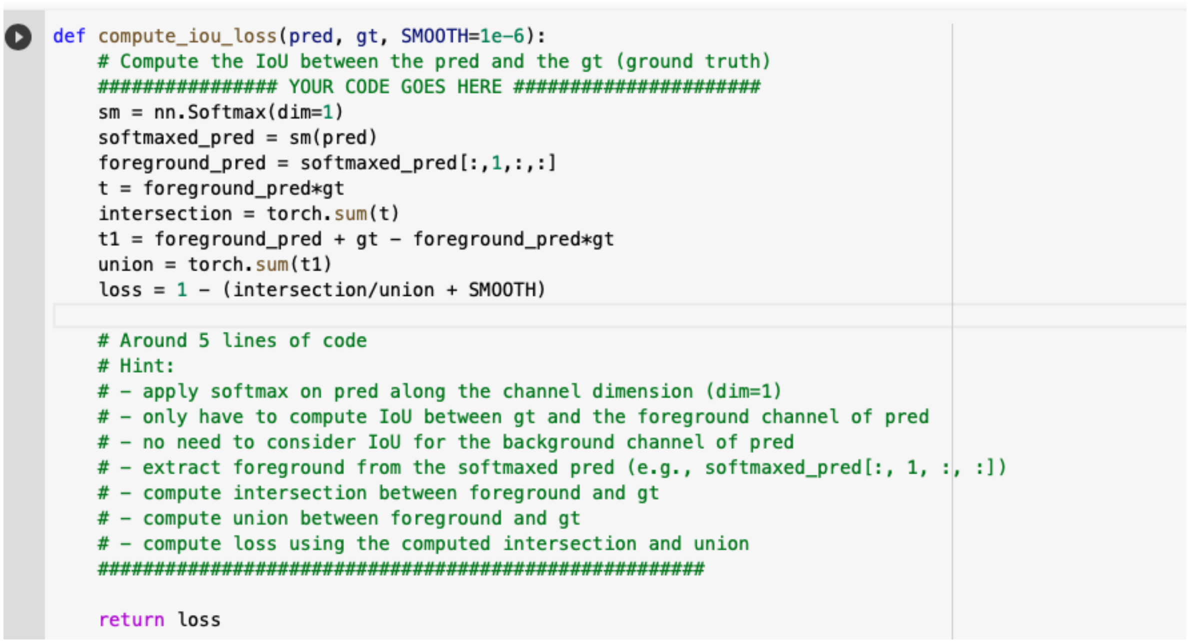
```
Best model achieves mIOU: 0.3500
```

Figure 9: Caption

5 Part D.2

5.1

Look at Fig 10 and Fig 11
Best validation mIOU: 0.3085. Best validation mIOU for this model (with IOU loss) is lower than the best validation mIOU for model with cross entropy loss.We get worse mIOU compared to the mIOU when training with cross entropy loss.

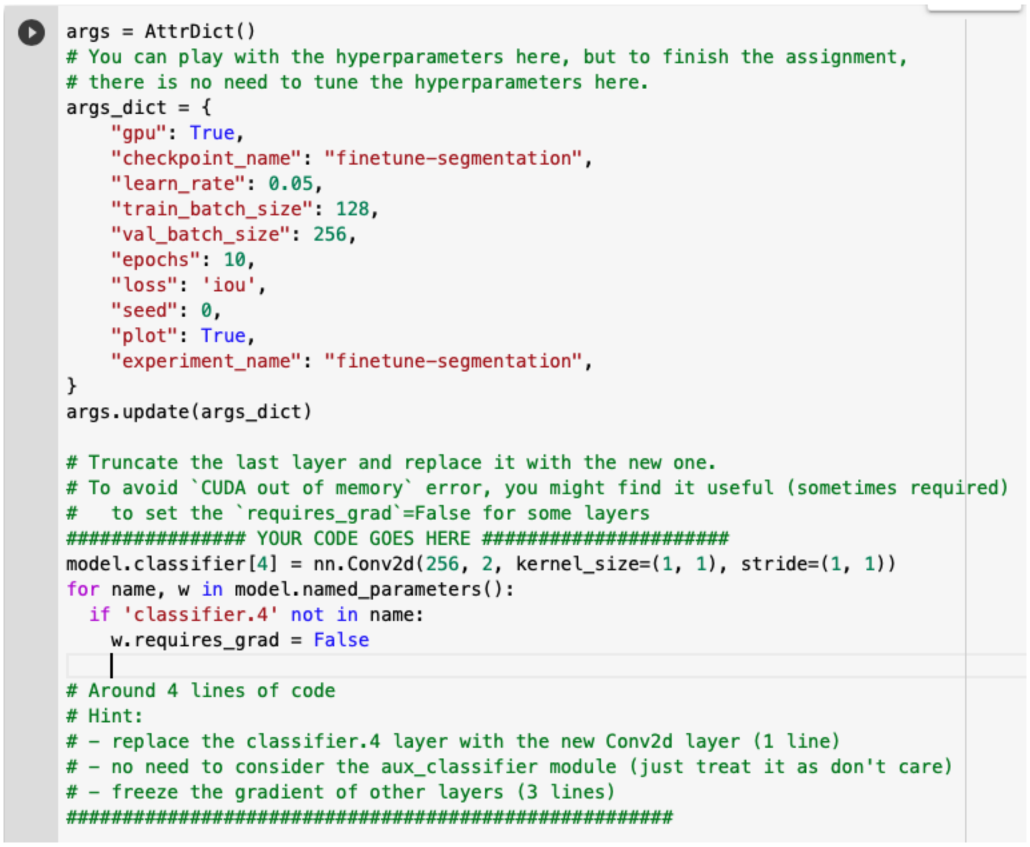


```
def compute_iou_loss(pred, gt, SMOOTH=1e-6):
    # Compute the IoU between the pred and the gt (ground truth)
    ################ YOUR CODE GOES HERE #####################
    sm = nn.Softmax(dim=1)
    softmaxed_pred = sm(pred)
    foreground_pred = softmaxed_pred[:,1,:,:]
    t = foreground_pred*gt
    intersection = torch.sum(t)
    t1 = foreground_pred + gt - foreground_pred*gt
    union = torch.sum(t1)
    loss = 1 - (intersection/union + SMOOTH)

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

    return loss
```

Figure 10:

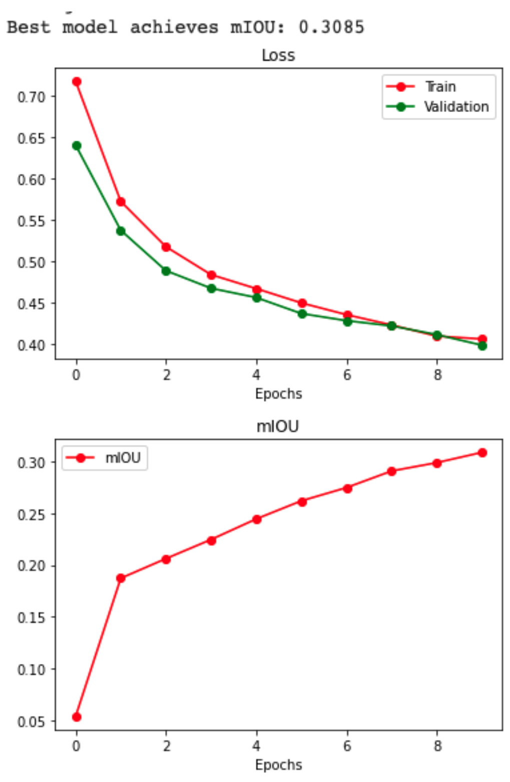
5.2

Look at Fig 12



```
args = AttrDict()
# You can play with the hyperparameters here, but to finish the assignment,
# there is no need to tune the hyperparameters here.
args_dict = {
    "gpu": True,
    "checkpoint_name": "finetune-segmentation",
    "learn_rate": 0.05,
    "train_batch_size": 128,
    "val_batch_size": 256,
    "epochs": 10,
    "loss": 'iou',
    "seed": 0,
    "plot": True,
    "experiment_name": "finetune-segmentation",
}
args.update(args_dict)

# 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 ######################
model.classifier[4] = nn.Conv2d(256, 2, kernel_size=(1, 1), stride=(1, 1))
for name, w in model.named_parameters():
  if 'classifier.4' not in name:
    w.requires_grad = False

# 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)
#######################################################
```

Figure 11:



```
Best model achieves mIOU: 0.3085
```

Figure 12: