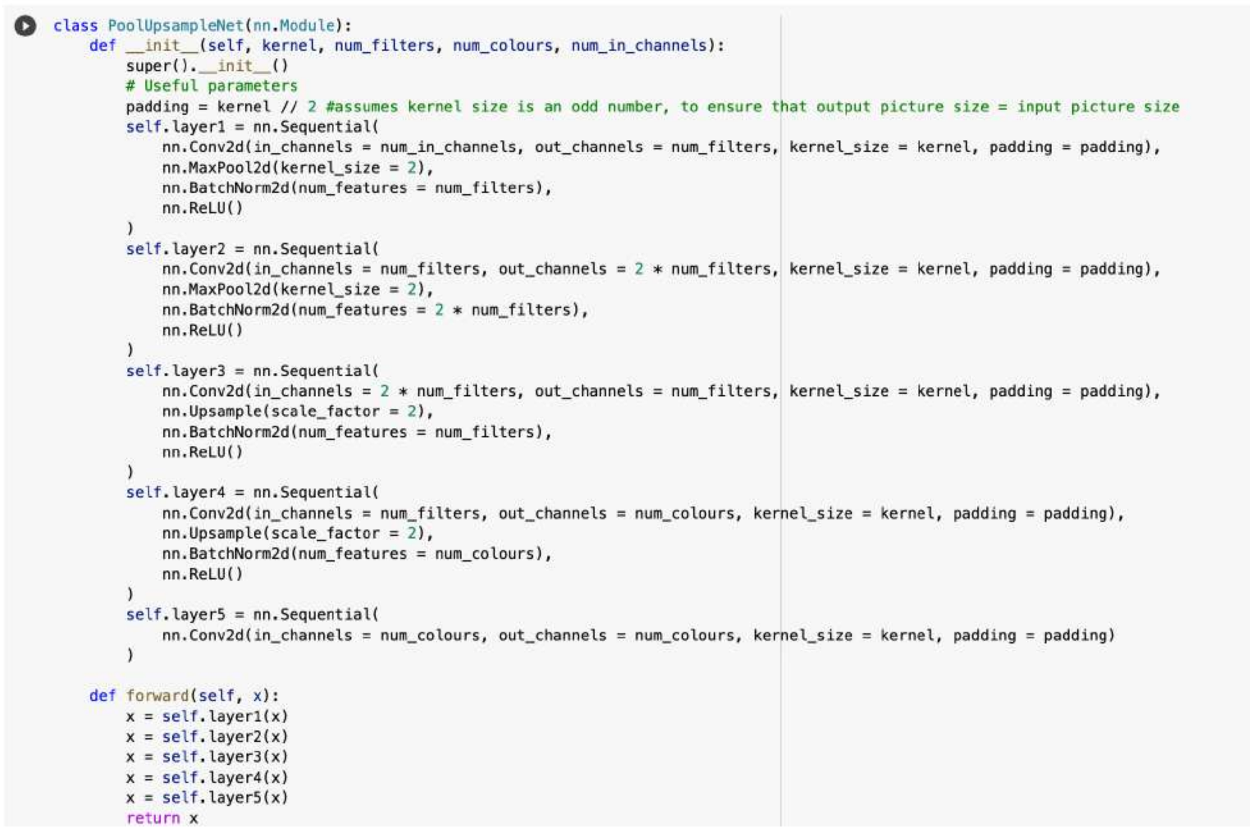
CSC413 2021W PA2 Write Up Guolun Li

Part A:

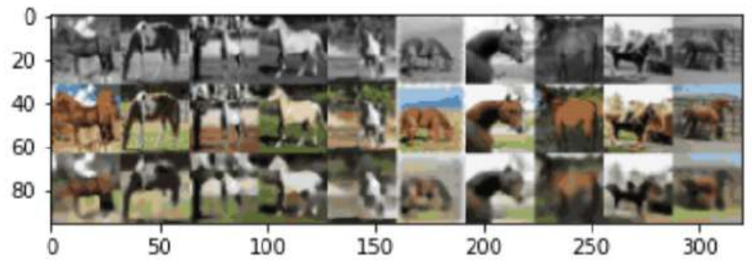
A1.

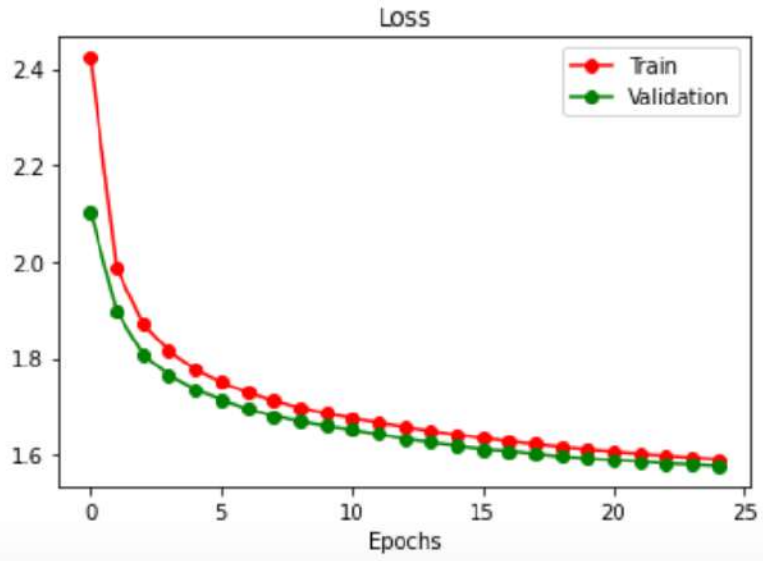


```
class PoolUpsampleNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()
        # Useful parameters
        padding = kernel // 2 #assumes kernel size is an odd number, to ensure that output picture size = input picture size
        self.layer1 = 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()
        )
        self.layer2 = 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()
        )
        self.layer3 = 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()
        )
        self.layer4 = 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()
        )
        self.layer5 = nn.Sequential(
            nn.Conv2d(in_channels = num_colours, out_channels = num_colours, kernel_size = kernel, padding = padding)
        )

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.layer5(x)
        return x
```

A2.

```
Epoch [22/25], Val Loss: 1.5858, Val Acc: 41.3%, Time(s): 28.26
Epoch [23/25], Loss: 1.5977, Time (s): 29
Epoch [23/25], Val Loss: 1.5814, Val Acc: 41.4%, Time(s): 29.92
Epoch [24/25], Loss: 1.5939, Time (s): 31
Epoch [24/25], Val Loss: 1.5791, Val Acc: 41.5%, Time(s): 31.61
Epoch [25/25], Loss: 1.5902, Time (s): 32
Epoch [25/25], Val Loss: 1.5760, Val Acc: 41.6%, Time(s): 33.31
```





41.6% accuracy means for each image, on average 40% of the area are colored correctly. The final result looks ok since this is our very first trial, but there's obviously room for improvement.

A3.

Assuming batch size to be 1, we derive general expressions for input of size (C,W,D, N) for each type of layers first. Here C = # of input channels, W and D are respectively the width and height of each input picture, and N = # of output channels.

We provide justification for # of connections for BatchNorm2d and UpSample layers:
The batchnorm layer computes C different expected values and variances over all possible W*D pixels. For each channel, W*D input pixels are fully connected to W*D output pixels. Thus total # of connection is $C * (WD)^2$.
The upsample layer is similar to max pooling layer, but in a reverse way. Each input pixel uniquely connects to 4 output pixels. Thus total # of connection is 4 * C * WD.

	Conv2d	MaxPool2d	BatchNorm2d	UpSample

# of weights	$CNk^2 + N$	0	$2C$	0
# of outputs	NWD	$C\frac{W}{2}\frac{D}{2}$	CWD	$C * 2W * 2D$
# of connections	k^2CNWD	CWD	CW^2D^2	$C * 2W * 2D$

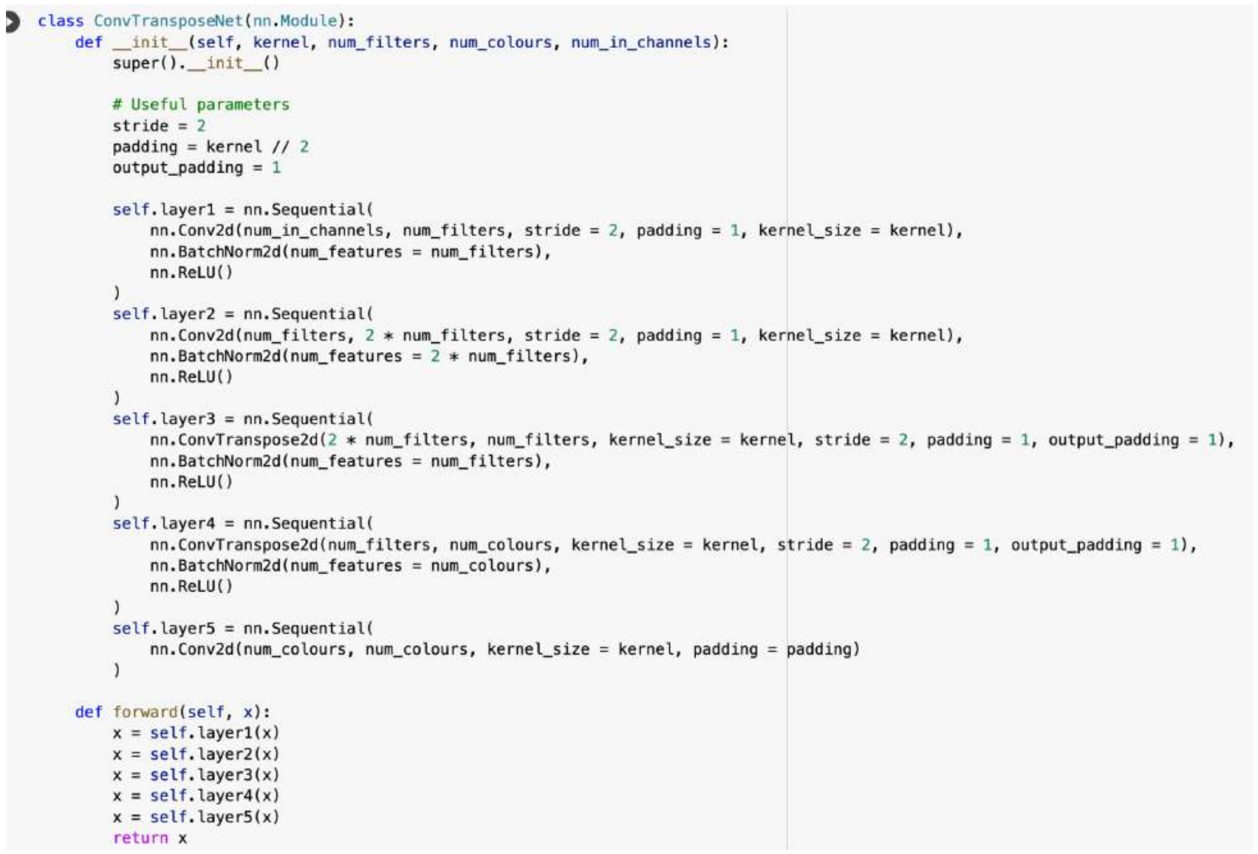
With initial image size of 32 x 32:

	# of Inputs	# of Weights	# of Outputs	# of connections
Conv1	(NIC, 32, 32, NF)	$NIC * NF * k^2 + NF$	$NF * 32 * 32$	$k^2NIC * NF * 32^2$
MaxPool1	(NF, 32, 32, NF)	0	$NF * 16 * 16$	$NF * 32^2$
BatchNorm1	(NF, 16, 16, NF)	2NF	$NF * 16 * 16$	$NF * 16^4$
Conv2	(NF, 16, 16, 2NF)	$2NF^2 * k^2 + 2NF$	$2NF * 16 * 16$	$k^2 * NF * 2NF * 16^2$
MaxPool2	(2NF, 16, 16, 2NF)	0	$2NF * 8 * 8$	$2NF * 16^2$
BatchNorm2	(2NF, 8, 8, 2NF)	4NF	$2NF * 8 * 8$	$2NF * 8^4$
Conv3	(2NF, 8, 8, NF)	$2NF * NF * k^2 + NF$	$NF * 8 * 8$	$k^2 * 2NF * NF * 8^2$
UpSample3	(NF,8,8,NF)	0	$NF * 16 * 16$	$NF * 16^2$
BatchNorm3	(NF,16,16,NF)	2NF	$NF * 16 * 16$	$NF * 16^4$
Conv4	(NF,16,16,NC)	$NF * NC * k^2 + NC$	$NC * 16 * 16$	$k^2 * NF * NC * 16^2$
UpSample4	(NC,16,16,NC)	0	$NC * 32 * 32$	$NC * 32^2$
BatchNorm4	(NC,32,32,NC)	2NC	$NC * 32 * 32$	$NC * 32^4$
Conv5	(NC,32,32,NC)	$NC^2 * k^2 + NC$	$NC * 32 * 32$	$k^2NC^2 32^2$

Total # of weights = $k^2(NIC * NF + 4NF^2 + NF * NC + NC^2) + 12NF + 4NC$
Total # of outputs = $2880NF + 3328NC$
Total # of connections = $k^2(1024NIC * NF + 640NF^2 + 256NF * NC + 1024NC^2) + 141056NF + 1280NC$

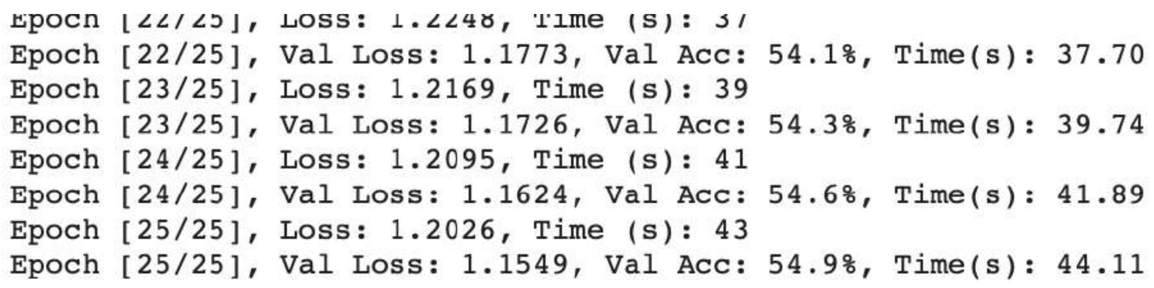
When weight and height are doubled, we can calculate the total #'s in the same way to obtain the following results:
Total # of weights = $k^2(NIC * NF + 4NF^2 + NF * NC + NC^2) + 12NF + 4NC$
Total # of outputs = $11520NF + 13312NC$
Total # of connections = $4k^2(1024NIC * NF + 640NF^2 + 256NF * NC + 1024NC^2) + 2235392NF + 16781312NC$

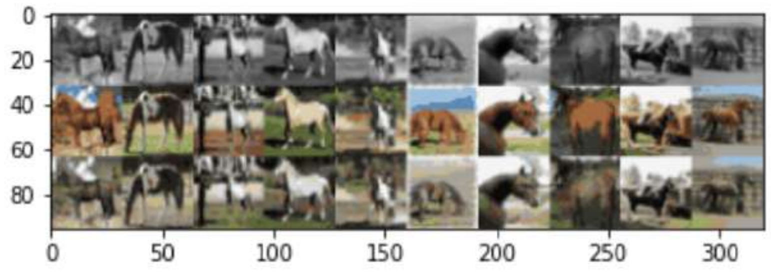
Part B:
B1:

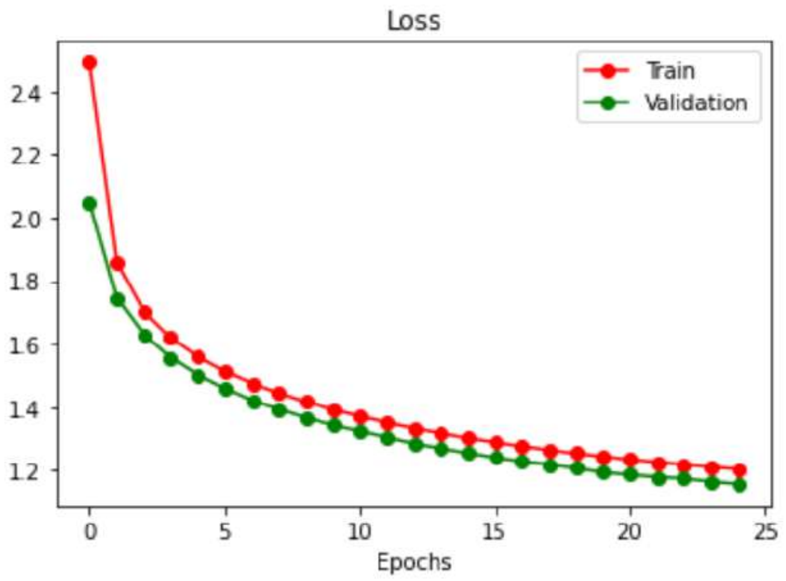


```python
class ConvTransposeNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

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

        self.layer1 = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, stride = 2, padding = 1, kernel_size = kernel),
            nn.BatchNorm2d(num_features = num_filters),
            nn.ReLU()
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(num_filters, 2 * num_filters, stride = 2, padding = 1, kernel_size = kernel),
            nn.BatchNorm2d(num_features = 2 * num_filters),
            nn.ReLU()
        )
        self.layer3 = nn.Sequential(
            nn.ConvTranspose2d(2 * num_filters, num_filters, kernel_size = kernel, stride = 2, padding = 1, output_padding = 1),
            nn.BatchNorm2d(num_features = num_filters),
            nn.ReLU()
        )
        self.layer4 = nn.Sequential(
            nn.ConvTranspose2d(num_filters, num_colours, kernel_size = kernel, stride = 2, padding = 1, output_padding = 1),
            nn.BatchNorm2d(num_features = num_colours),
            nn.ReLU()
        )
        self.layer5 = nn.Sequential(
            nn.Conv2d(num_colours, num_colours, kernel_size = kernel, padding = padding)
        )

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.layer5(x)
        return x
```

B2:


```
Epoch [22/25], Loss: 1.2248, Time (s): 37
Epoch [22/25], Val Loss: 1.1773, Val Acc: 54.1%, Time(s): 37.70
Epoch [23/25], Loss: 1.2169, Time (s): 39
Epoch [23/25], Val Loss: 1.1726, Val Acc: 54.3%, Time(s): 39.74
Epoch [24/25], Loss: 1.2095, Time (s): 41
Epoch [24/25], Val Loss: 1.1624, Val Acc: 54.6%, Time(s): 41.89
Epoch [25/25], Loss: 1.2026, Time (s): 43
Epoch [25/25], Val Loss: 1.1549, Val Acc: 54.9%, Time(s): 44.11
```



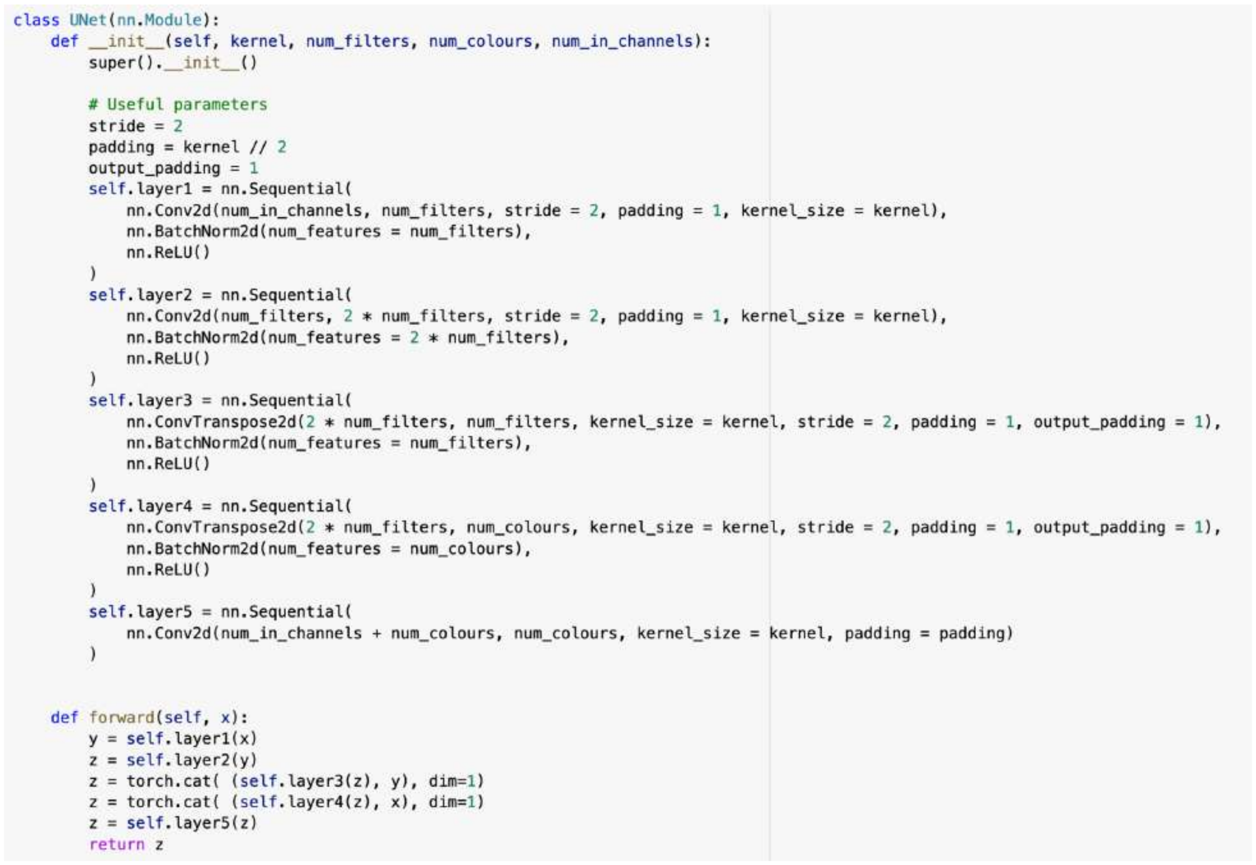


B3.
Compared to the previous result, the validation loss decreased from 1.576 to 1.1549(26.7% relative decrement) and the validation accuracy increased from 41.6% to 54.9%(13.3% absolute increment). This is a huge boost in model performance.
Because in a transposed convolution layer, multiple input channels are attached to one output channel, whereas in a upsampling layer, one input channel can only connect to one output channel. As a result, transposed convolutional layer provides more learnable parameters than UpSampling layer for the model to lear to upsample optimally.
B4.
See table below.

Layer	parameter	K = 3	K = 4	K=5
Conv2d	Padding	P = 1	P = 1	P = 2
ConvTrans2d	Padding	P = 1	P = 2	P = 2

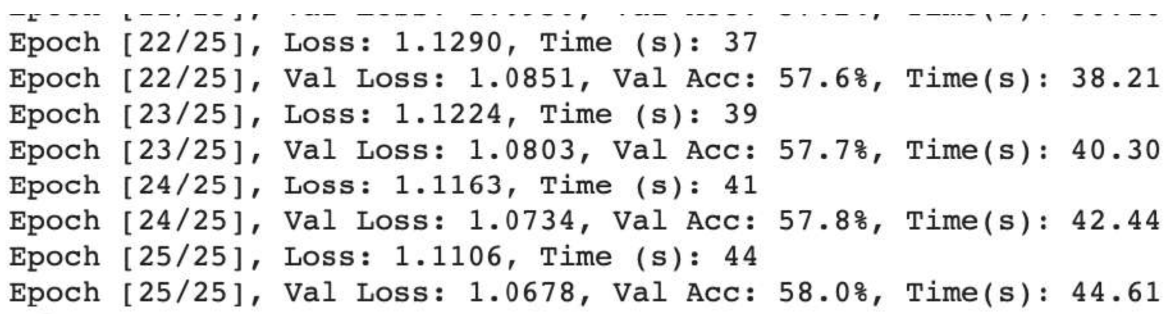
	Output padding	$P_{out} = 1$	$P_{out} = 2$	$P_{out} = 1$

B5.
Smaller the batch size, better the validation loss and output image quality.

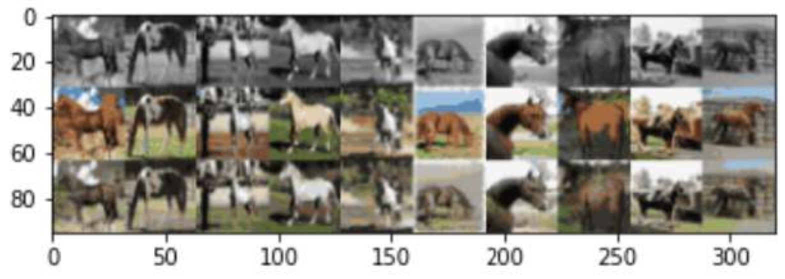
Part C:
C1:


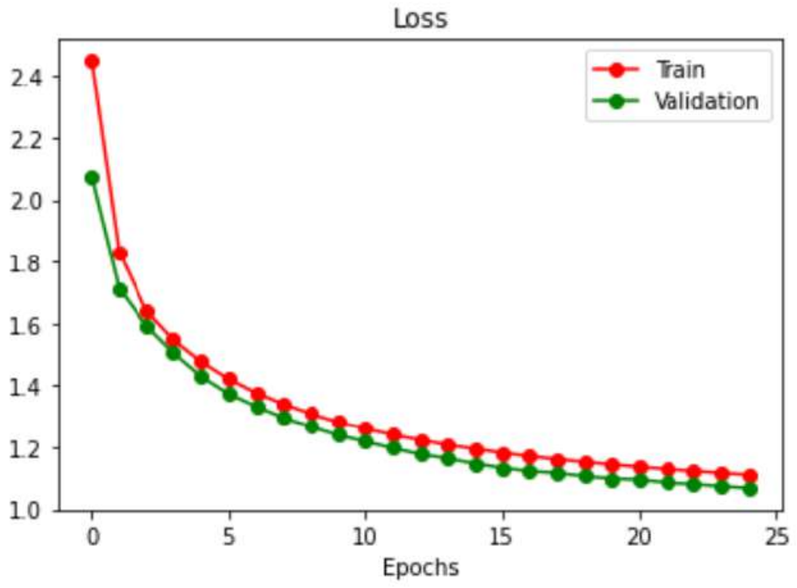
```python
class UNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        stride = 2
        padding = kernel // 2
        output_padding = 1
        self.layer1 = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, stride = 2, padding = 1, kernel_size = kernel),
            nn.BatchNorm2d(num_features = num_filters),
            nn.ReLU()
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(num_filters, 2 * num_filters, stride = 2, padding = 1, kernel_size = kernel),
            nn.BatchNorm2d(num_features = 2 * num_filters),
            nn.ReLU()
        )
        self.layer3 = nn.Sequential(
            nn.ConvTranspose2d(2 * num_filters, num_filters, kernel_size = kernel, stride = 2, padding = 1, output_padding = 1),
            nn.BatchNorm2d(num_features = num_filters),
            nn.ReLU()
        )
        self.layer4 = nn.Sequential(
            nn.ConvTranspose2d(2 * num_filters, num_colours, kernel_size = kernel, stride = 2, padding = 1, output_padding = 1),
            nn.BatchNorm2d(num_features = num_colours),
            nn.ReLU()
        )
        self.layer5 = nn.Sequential(
            nn.Conv2d(num_in_channels + num_colours, num_colours, kernel_size = kernel, padding = padding)
        )


    def forward(self, x):
        y = self.layer1(x)
        z = self.layer2(y)
        z = torch.cat( (self.layer3(z), y), dim=1)
        z = torch.cat( (self.layer4(z), x), dim=1)
        z = self.layer5(z)
        return z
```

C2:



```
Epoch [22/25], Loss: 1.1290, Time (s): 37
Epoch [22/25], Val Loss: 1.0851, Val Acc: 57.6%, Time(s): 38.21
Epoch [23/25], Loss: 1.1224, Time (s): 39
Epoch [23/25], Val Loss: 1.0803, Val Acc: 57.7%, Time(s): 40.30
Epoch [24/25], Loss: 1.1163, Time (s): 41
Epoch [24/25], Val Loss: 1.0734, Val Acc: 57.8%, Time(s): 42.44
Epoch [25/25], Loss: 1.1106, Time (s): 44
Epoch [25/25], Val Loss: 1.0678, Val Acc: 58.0%, Time(s): 44.61
```

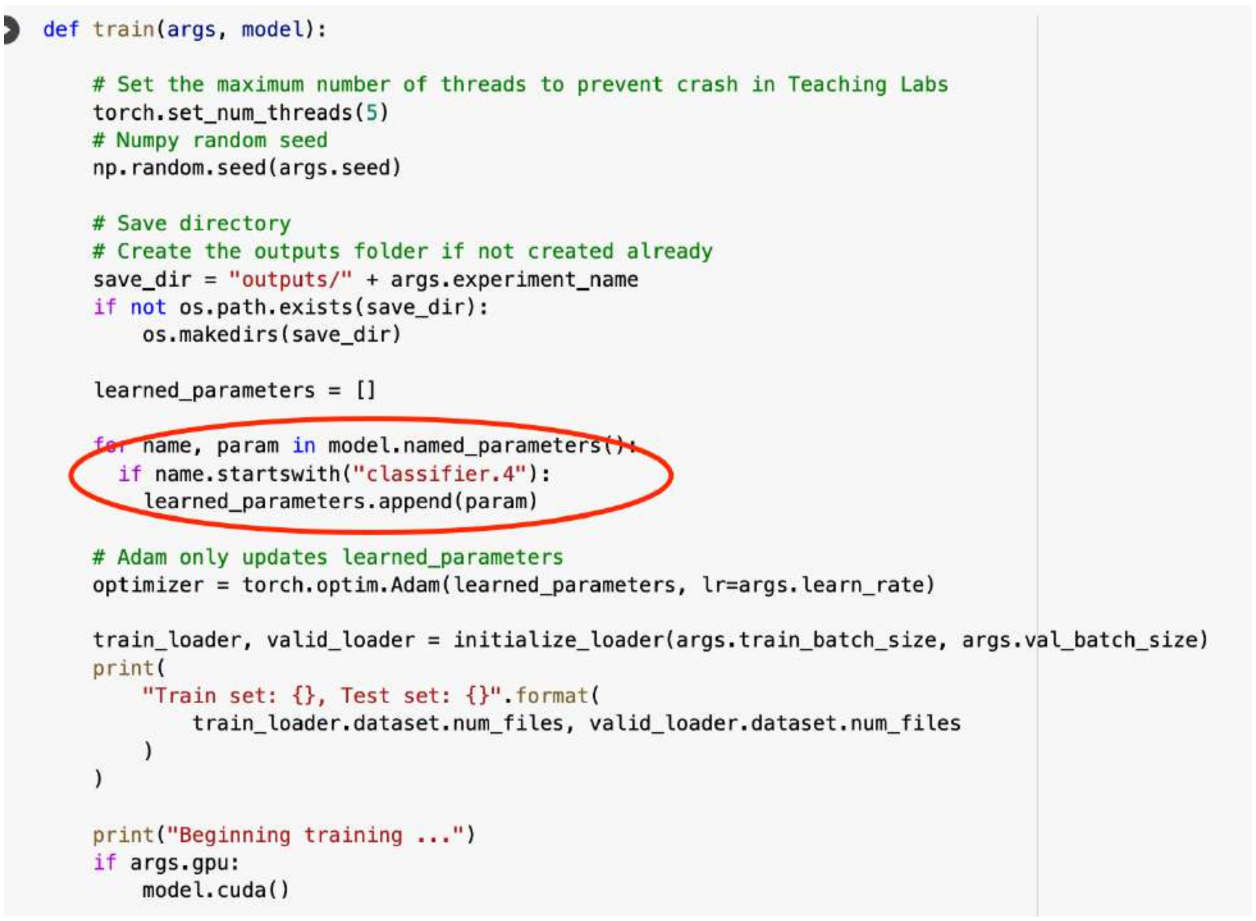




C3:
1)The validation loss decreased from 1.1549 to 1.0678(7.5% relative decrement), and the validation accuracy increased from 54.9% to 58.0%(3.1% absolute increment). The model is slightly better than the previous one.
2) Skip connections indeed improve the validation loss and accuracy.
3) Such improvement is not large enough to be considered “qualitative”.
4) There are two reasons why skipnet improve performance:
--Skipnet adds additional trainable parameters to the model. Here we’re using validation loss as a metric, so the model performance literally improves with model complexity.
--Without skipped connections, only the most essential information is retained. By adding skip connections, the model is able to make final decision based on details inside the image.

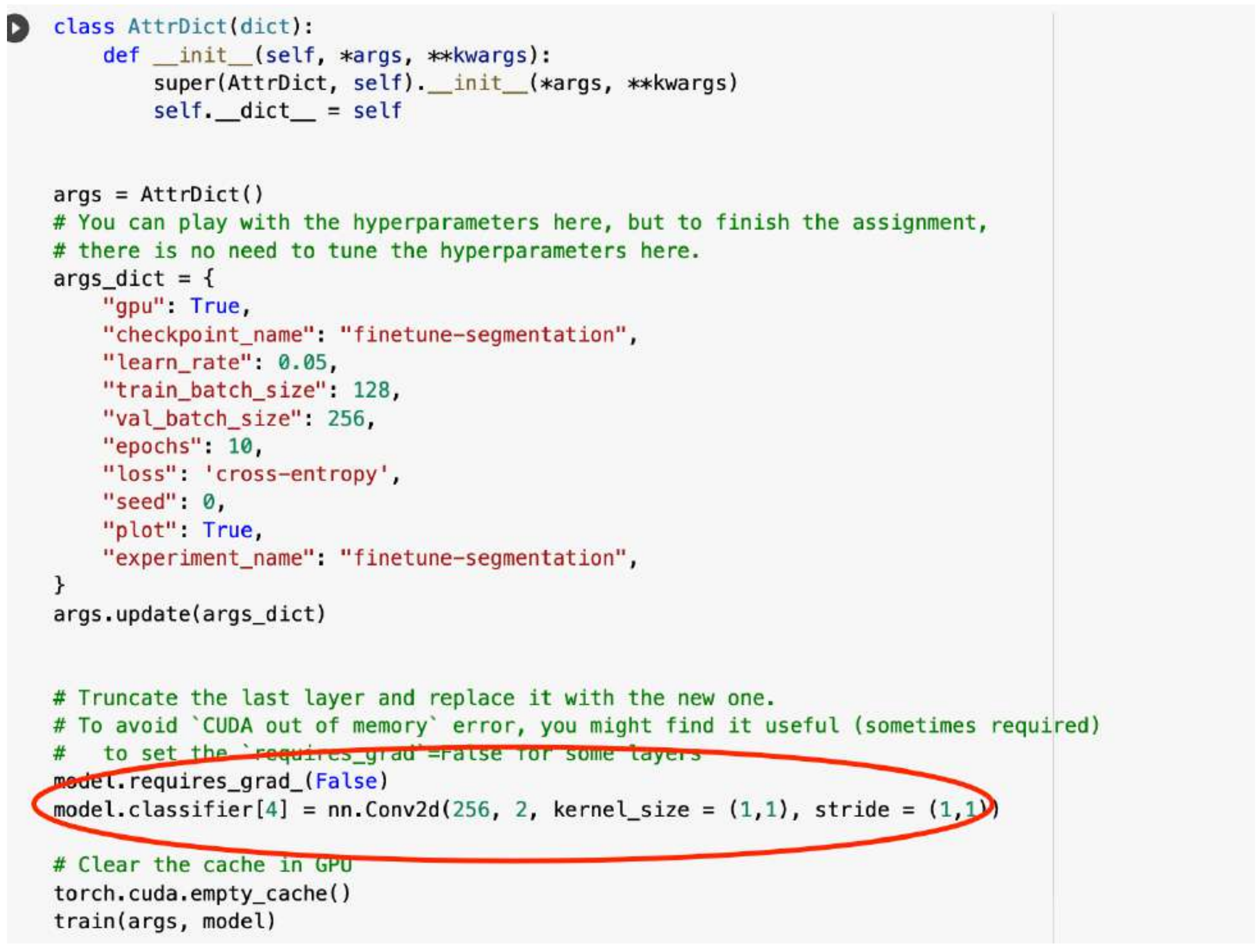
Part D.1:

D.1.1:

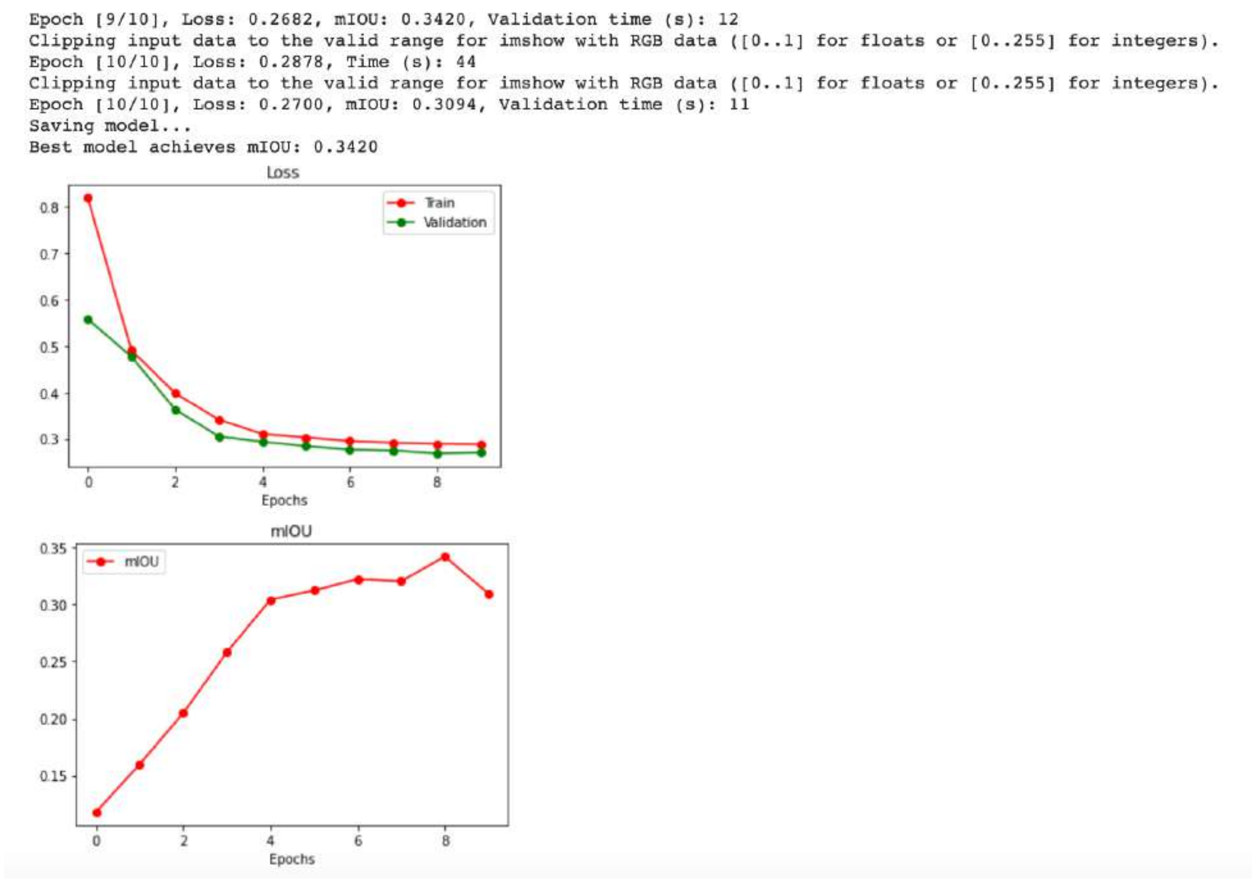


```
def train(args, model):

    # 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 = []

    for name, param in model.named_parameters():
      if name.startswith("classifier.4"):
        learned_parameters.append(param)

    # Adam only updates learned_parameters
    optimizer = torch.optim.Adam(learned_parameters, lr=args.learn_rate)

    train_loader, valid_loader = initialize_loader(args.train_batch_size, args.val_batch_size)
    print(
        "Train set: {}, Test set: {}".format(
            train_loader.dataset.num_files, valid_loader.dataset.num_files
        )
    )

    print("Beginning training ...")
    if args.gpu:
        model.cuda()
```

D.1.2:



```python
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
model.requires_grad_(False)
model.classifier[4] = nn.Conv2d(256, 2, kernel_size = (1,1), stride = (1,1))

# Clear the cache in GPU
torch.cuda.empty_cache()
train(args, model)
```

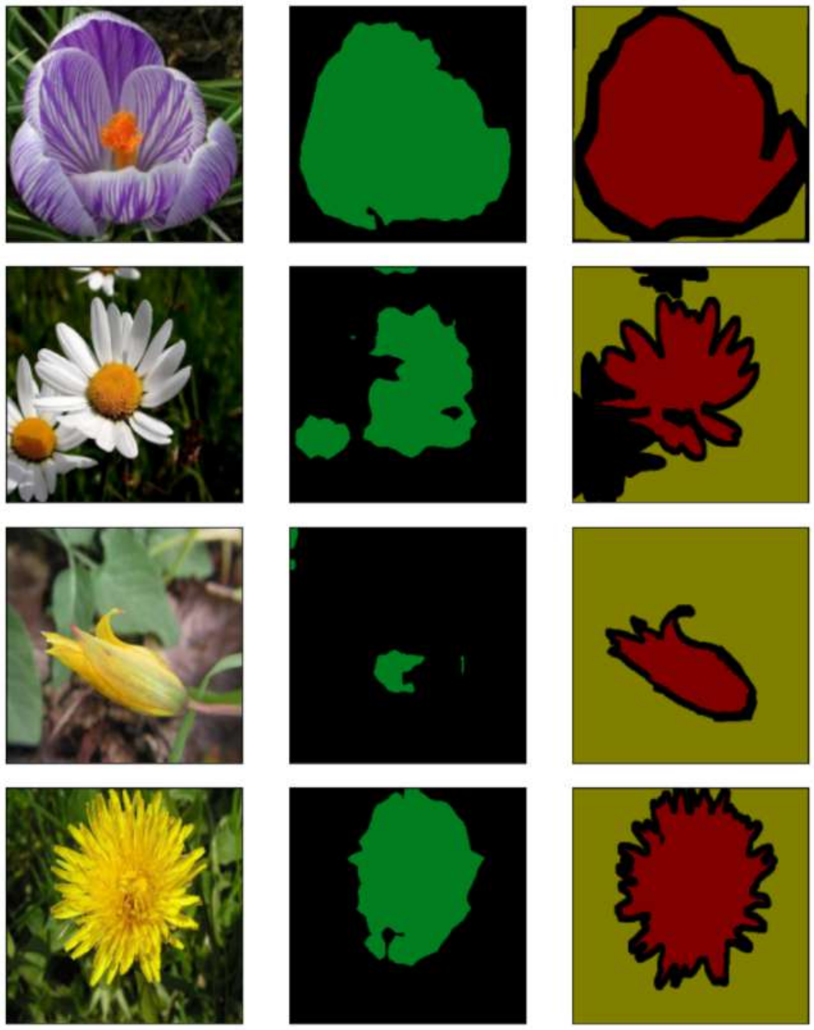


```
Epoch [9/10], Loss: 0.2682, mIOU: 0.3420, Validation time (s): 12
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Epoch [10/10], Loss: 0.2878, Time (s): 44
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Epoch [10/10], Loss: 0.2700, mIOU: 0.3094, Validation time (s): 11
Saving model...
Best model achieves mIOU: 0.3420
```

The best mIOU is 0.3420.

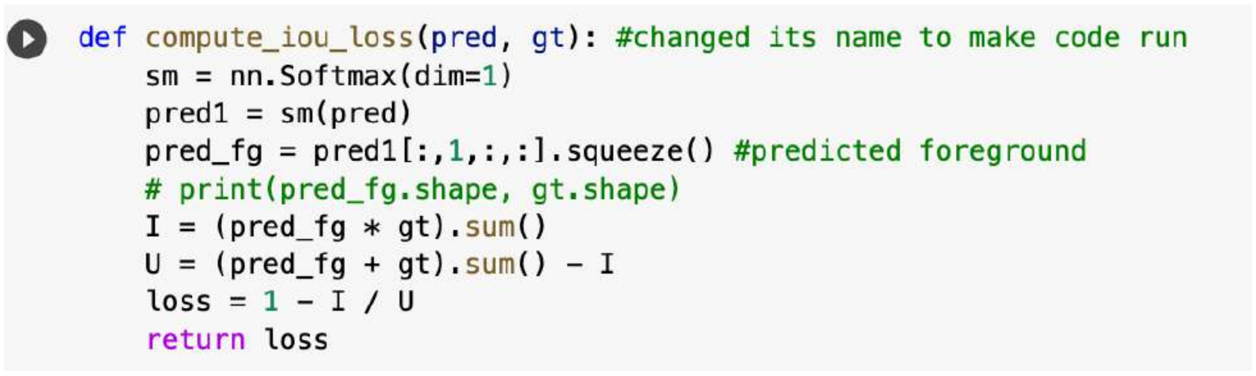
D.1.3:

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
```



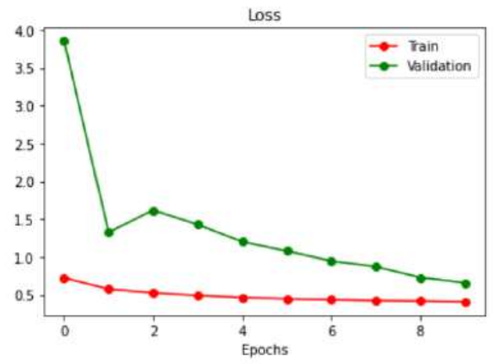
Part D.2:

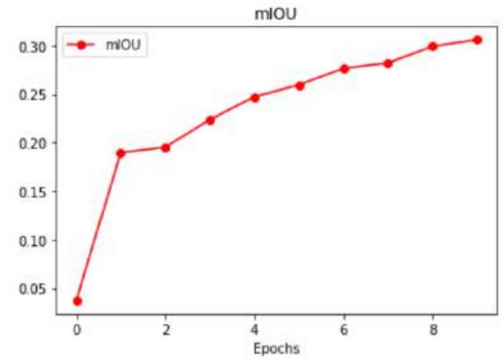
D.2.1:



```
def compute_iou_loss(pred, gt): #changed its name to make code run
    sm = nn.Softmax(dim=1)
    pred1 = sm(pred)
    pred_fg = pred1[:,1,:,:].squeeze() #predicted foreground
    # print(pred_fg.shape, gt.shape)
    I = (pred_fg * gt).sum()
    U = (pred_fg + gt).sum() - I
    loss = 1 - I / U
    return loss
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Epoch [10/10], Loss: 0.4075, Time (s): 44
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Epoch [10/10], Loss: 0.6572, mIOU: 0.3061, Validation time (s): 12
Saving model...
Best model achieves mIOU: 0.3061
```

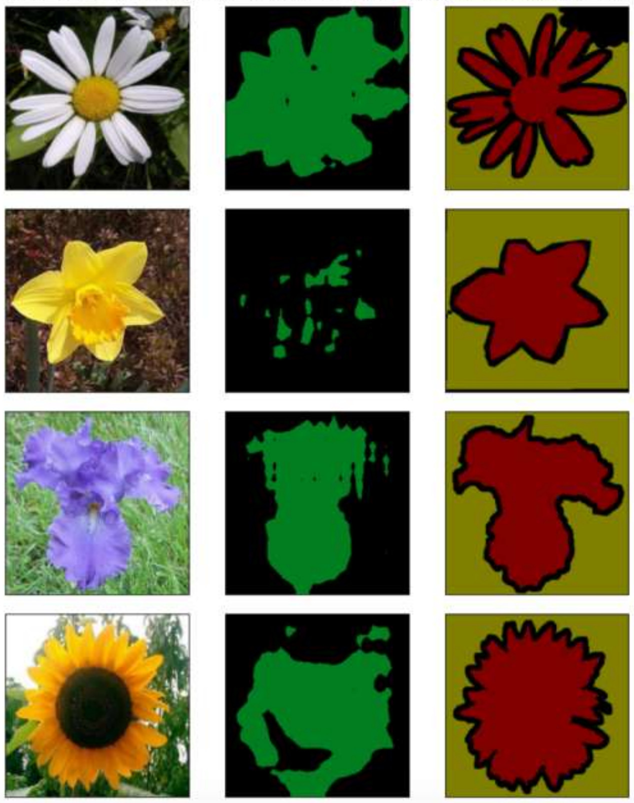




The best mIOU is now 0.3061, which is worse than the previous model.

D.2.2:

```
Clipping input data to the valid range for imshow with RGB data ([0..1] f
```



```
Clipping input data to the valid range for imshow with RGB data ([0..1]
```