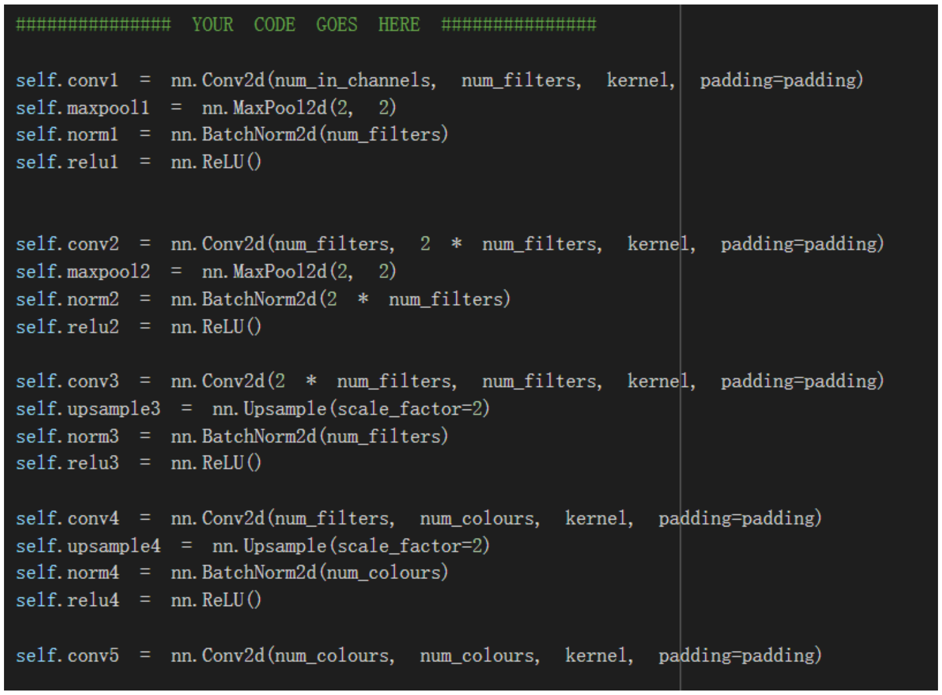
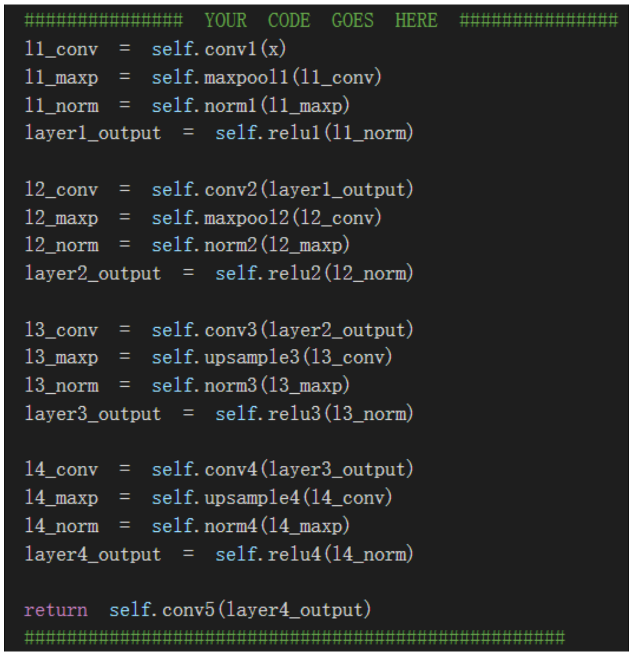
Part A:

Code for initialization:



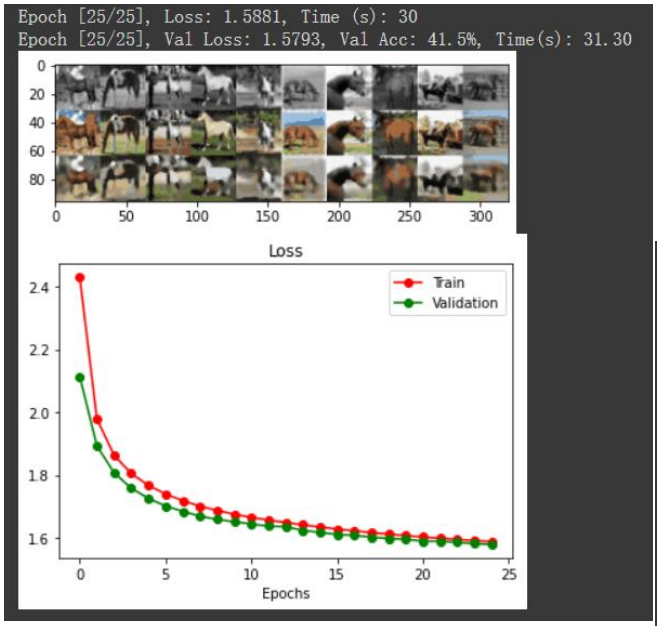
```
################ YOUR CODE GOES HERE ################

self.conv1 = nn.Conv2d(num_in_channels, num_filters, kernel, padding=padding)
self.maxpool1 = nn.MaxPool2d(2, 2)
self.norm1 = nn.BatchNorm2d(num_filters)
self.relu1 = nn.ReLU()


self.conv2 = nn.Conv2d(num_filters, 2 * num_filters, kernel, padding=padding)
self.maxpool2 = nn.MaxPool2d(2, 2)
self.norm2 = nn.BatchNorm2d(2 * num_filters)
self.relu2 = nn.ReLU()

self.conv3 = nn.Conv2d(2 * num_filters, num_filters, kernel, padding=padding)
self.upsample3 = nn.Upsample(scale_factor=2)
self.norm3 = nn.BatchNorm2d(num_filters)
self.relu3 = nn.ReLU()

self.conv4 = nn.Conv2d(num_filters, num_colours, kernel, padding=padding)
self.upsample4 = nn.Upsample(scale_factor=2)
self.norm4 = nn.BatchNorm2d(num_colours)
self.relu4 = nn.ReLU()

self.conv5 = nn.Conv2d(num_colours, num_colours, kernel, padding=padding)
```

Code for forward_pass:



```
################ YOUR CODE GOES HERE ################
l1_conv = self.conv1(x)
l1_maxp = self.maxpool1(l1_conv)
l1_norm = self.norm1(l1_maxp)
layer1_output = self.relu1(l1_norm)

l2_conv = self.conv2(layer1_output)
l2_maxp = self.maxpool2(l2_conv)
l2_norm = self.norm2(l2_maxp)
layer2_output = self.relu2(l2_norm)

l3_conv = self.conv3(layer2_output)
l3_maxp = self.upsample3(l3_conv)
l3_norm = self.norm3(l3_maxp)
layer3_output = self.relu3(l3_norm)

l4_conv = self.conv4(layer3_output)
l4_maxp = self.upsample4(l4_conv)
l4_norm = self.norm4(l4_maxp)
layer4_output = self.relu4(l4_norm)

return self.conv5(layer4_output)
#####################################################
```

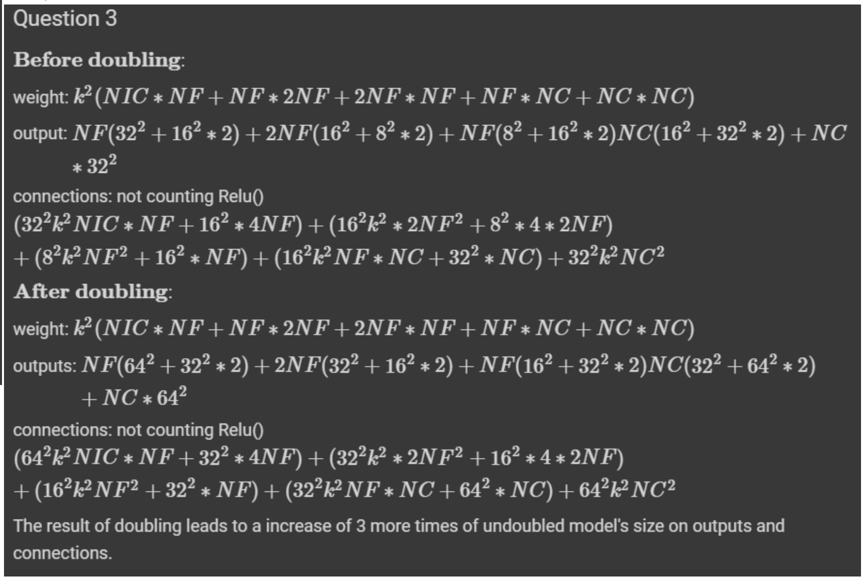
Result of training:



```
Epoch [25/25], Loss: 1.5881, Time (s): 30
Epoch [25/25], Val Loss: 1.5793, Val Acc: 41.5%, Time(s): 31.30
```

Comment:
The result seems not that satisfactory as the validation loss is high and the quality of image isn't looking good.

Question 3 Calculation result:



Question 3

Before doubling:

weight: $k^2(NIC*NF+NF*2NF+2NF*NF+NF*NC+NC*NC)$

output: $NF(32^2+16^2*2)+2NF(16^2+8^2*2)+NF(8^2+16^2*2)NC(16^2+32^2*2)+NC*32^2$

connections: not counting Relu()

$$(32^2k^2NIC*NF+16^2*4NF)+(16^2k^2*2NF^2+8^2*4*2NF)+(8^2k^2NF^2+16^2*NF)+(16^2k^2NF*NC+32^2*NC)+32^2k^2NC^2$$

After doubling:

weight: $k^2(NIC*NF+NF*2NF+2NF*NF+NF*NC+NC*NC)$

outputs: $NF(64^2+32^2*2)+2NF(32^2+16^2*2)+NF(16^2+32^2*2)NC(32^2+64^2*2)+NC*64^2$

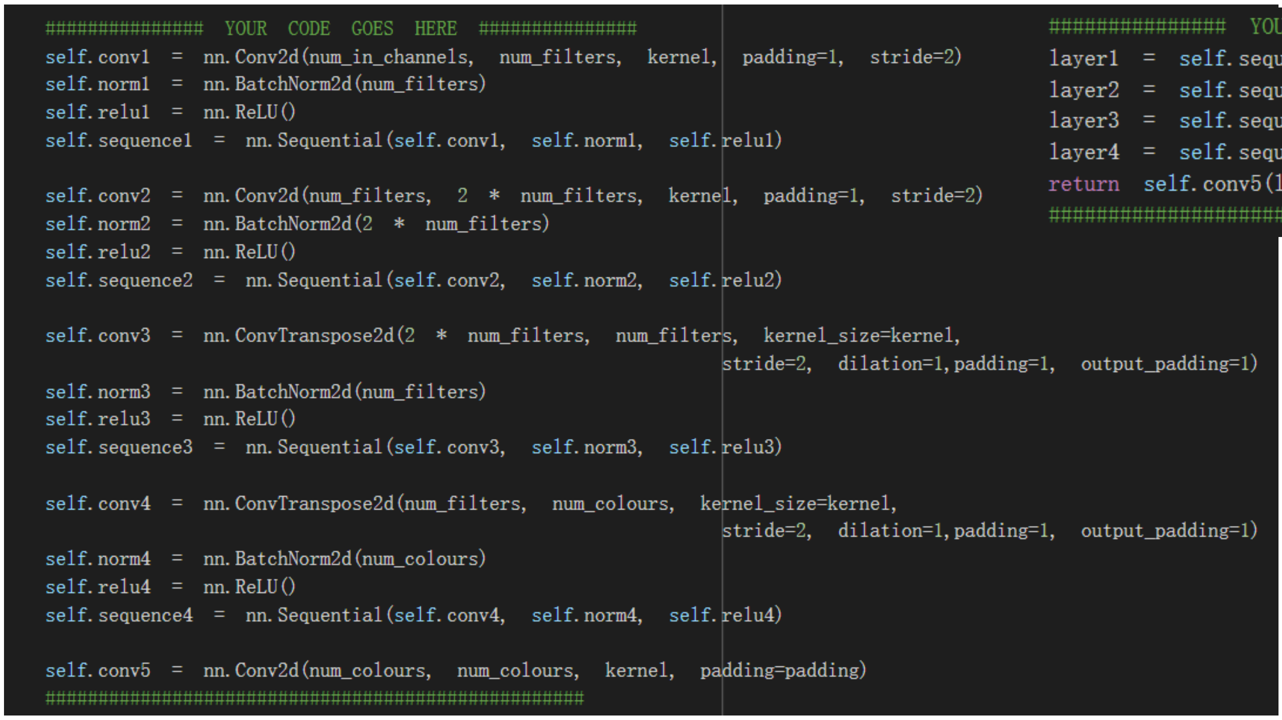
connections: not counting Relu()

$$(64^2k^2NIC*NF+32^2*4NF)+(32^2k^2*2NF^2+16^2*4*2NF)+(16^2k^2NF^2+32^2*NF)+(32^2k^2NF*NC+64^2*NC)+64^2k^2NC^2$$

The result of doubling leads to a increase of 3 more times of undoubled model's size on outputs and connections.

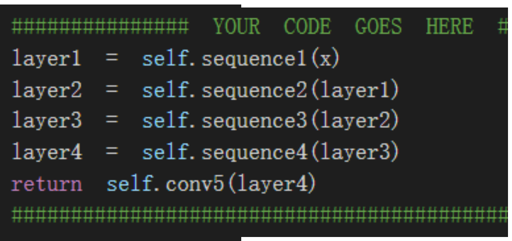
Part B:

Code for initialization:



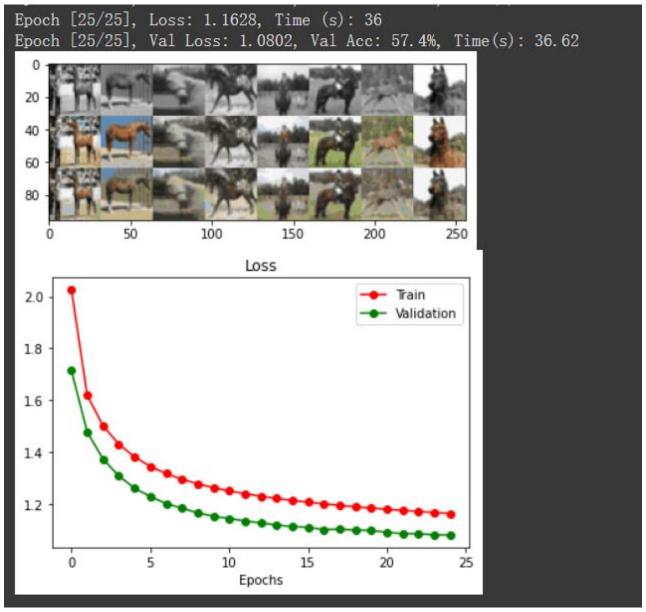
```
################ YOUR CODE GOES HERE ################
self.conv1 = nn.Conv2d(num_in_channels, num_filters, kernel, padding=1, stride=2)
self.norm1 = nn.BatchNorm2d(num_filters)
self.relu1 = nn.ReLU()
self.sequence1 = nn.Sequential(self.conv1, self.norm1, self.relu1)

self.conv2 = nn.Conv2d(num_filters, 2 * num_filters, kernel, padding=1, stride=2)
self.norm2 = nn.BatchNorm2d(2 * num_filters)
self.relu2 = nn.ReLU()
self.sequence2 = nn.Sequential(self.conv2, self.norm2, self.relu2)

self.conv3 = nn.ConvTranspose2d(2 * num_filters, num_filters, kernel_size=kernel,
                                stride=2, dilation=1,padding=1, output_padding=1)
self.norm3 = nn.BatchNorm2d(num_filters)
self.relu3 = nn.ReLU()
self.sequence3 = nn.Sequential(self.conv3, self.norm3, self.relu3)

self.conv4 = nn.ConvTranspose2d(num_filters, num_colours, kernel_size=kernel,
                                stride=2, dilation=1,padding=1, output_padding=1)
self.norm4 = nn.BatchNorm2d(num_colours)
self.relu4 = nn.ReLU()
self.sequence4 = nn.Sequential(self.conv4, self.norm4, self.relu4)

self.conv5 = nn.Conv2d(num_colours, num_colours, kernel, padding=padding)
######################################################
```

Code for training:



```
################ YOUR CODE GOES HERE #
layer1 = self.sequence1(x)
layer2 = self.sequence2(layer1)
layer3 = self.sequence3(layer2)
layer4 = self.sequence4(layer3)
return self.conv5(layer4)
#######################################
```

Result of training:



```
Epoch [25/25], Loss: 1.1628, Time (s): 36
Epoch [25/25], Val Loss: 1.0802, Val Acc: 57.4%, Time(s): 36.62
```

Answers for Question 3 – 5:

Question 3:

Compared to the results in part A, the training has improved quite a bit.
The validation error has also decreased from 1.6 in PoolUpsampleNet.
The Possible reason involves: max pooling layer's size is too large which leads to loss of features on image due to too much generalization.

Question 4:

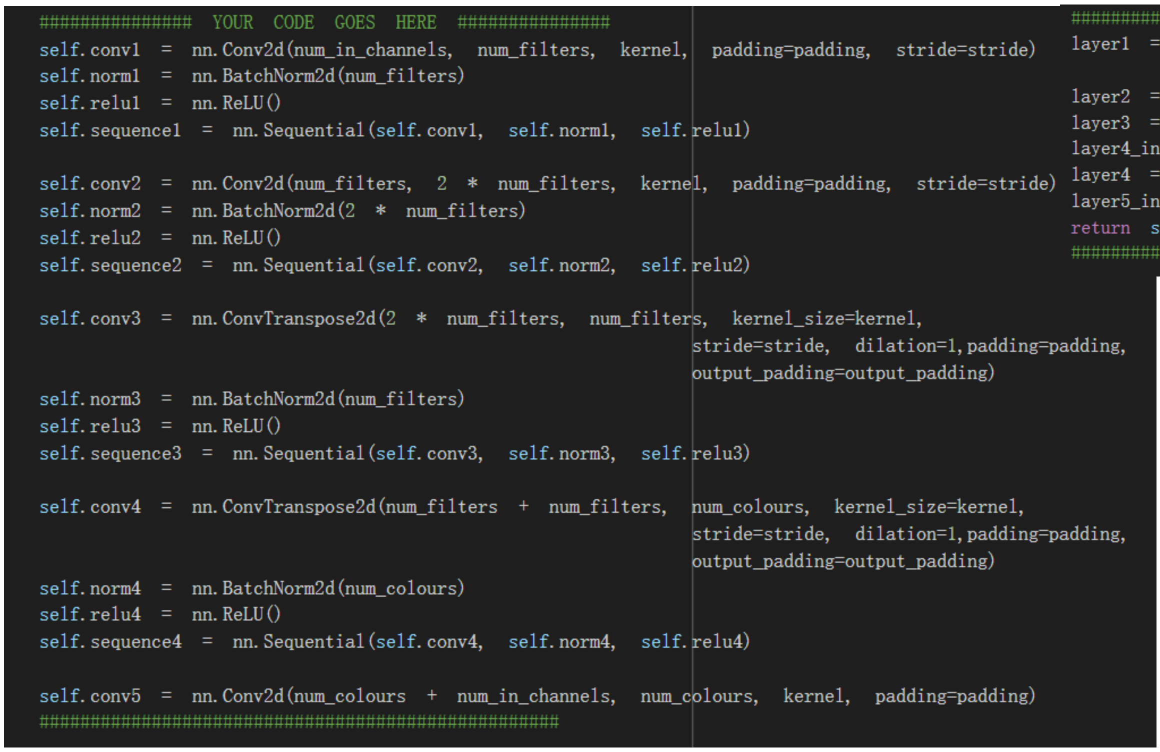
When kernel is 4: padding for first two convolution net is still 1 (given stride not change), and by refering to the man-page for nn.ConvTranspose2d, the choice for padding and output_padding is respectively: 2, 2.
When kernel is 5: padding for first two convolution net has to be 2 (given stride not change), and by refering to the man-page for nn.ConvTranspose2d, the choice for padding and output_padding is respectively: 3, 3.

Question 5:

By running several training examples, the validation loss decreases and quality of image output becomes better as batch size increases.

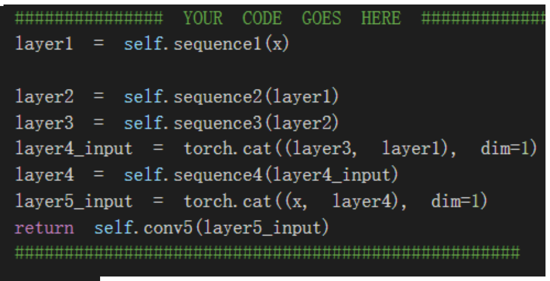
Part C:

Code for Initialization:



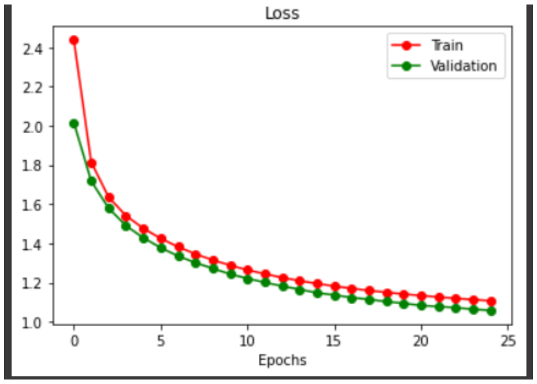
```
################ YOUR CODE GOES HERE ################
self.conv1 = nn.Conv2d(num_in_channels, num_filters, kernel, padding=padding, stride=stride)
self.norm1 = nn.BatchNorm2d(num_filters)
self.relu1 = nn.ReLU()
self.sequence1 = nn.Sequential(self.conv1, self.norm1, self.relu1)

self.conv2 = nn.Conv2d(num_filters, 2 * num_filters, kernel, padding=padding, stride=stride)
self.norm2 = nn.BatchNorm2d(2 * num_filters)
self.relu2 = nn.ReLU()
self.sequence2 = nn.Sequential(self.conv2, self.norm2, self.relu2)

self.conv3 = nn.ConvTranspose2d(2 * num_filters, num_filters, kernel_size=kernel,
                                                               stride=stride, dilation=1, padding=padding,
                                                               output_padding=output_padding)
self.norm3 = nn.BatchNorm2d(num_filters)
self.relu3 = nn.ReLU()
self.sequence3 = nn.Sequential(self.conv3, self.norm3, self.relu3)

self.conv4 = nn.ConvTranspose2d(num_filters + num_filters, num_colours, kernel_size=kernel,
                                                               stride=stride, dilation=1, padding=padding,
                                                               output_padding=output_padding)
self.norm4 = nn.BatchNorm2d(num_colours)
self.relu4 = nn.ReLU()
self.sequence4 = nn.Sequential(self.conv4, self.norm4, self.relu4)

self.conv5 = nn.Conv2d(num_colours + num_in_channels, num_colours, kernel, padding=padding)
###################################################
```

Code for Forward Pass:



```
############### YOUR CODE GOES HERE ############
layer1 = self.sequence1(x)

layer2 = self.sequence2(layer1)
layer3 = self.sequence3(layer2)
layer4_input = torch.cat((layer3, layer1), dim=1)
layer4 = self.sequence4(layer4_input)
layer5_input = torch.cat((x, layer4), dim=1)
return self.conv5(layer5_input)
##################################################
```

Result of training:



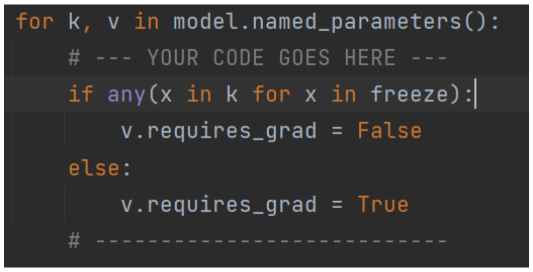
Answer for Question 3:

Both the training loss and validation loss have decreased. Final validation accuracy is also higher than previous two models. Output quality is also higher. Possible reasons include:

Previous output layers can be updated earlier during backward pass, improves effect of gradient descent.
At the same time by taking previous output together with current layer's output, the missed features could be restored.

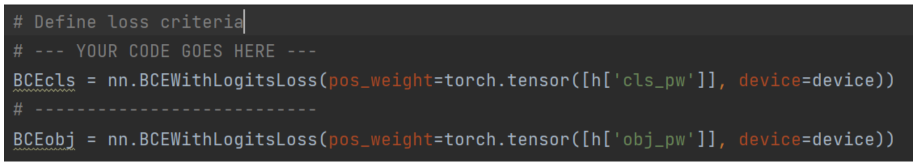
Part D:

Code for freezing layers:



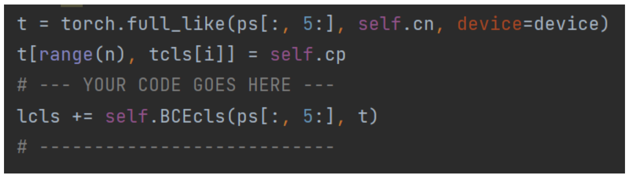
```
for k, v in model.named_parameters():
    # --- YOUR CODE GOES HERE ---
    if any(x in k for x in freeze):
        v.requires_grad = False
    else:
        v.requires_grad = True
    # --------------------------
```

Code for loss:

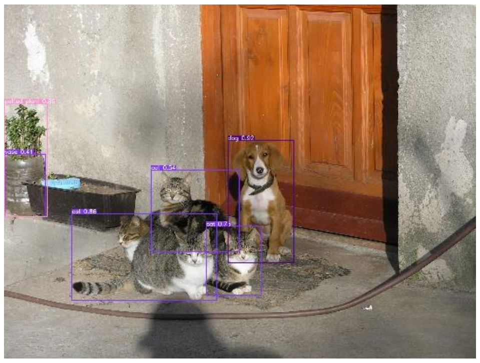


```
# Define loss criteria
# --- YOUR CODE GOES HERE ---
BCEcls = nn.BCEWithLogitsLoss(pos_weight=torch.tensor([h['cls_pw']], device=device))
# --------------------------
BCEobj = nn.BCEWithLogitsLoss(pos_weight=torch.tensor([h['obj_pw']], device=device))
```

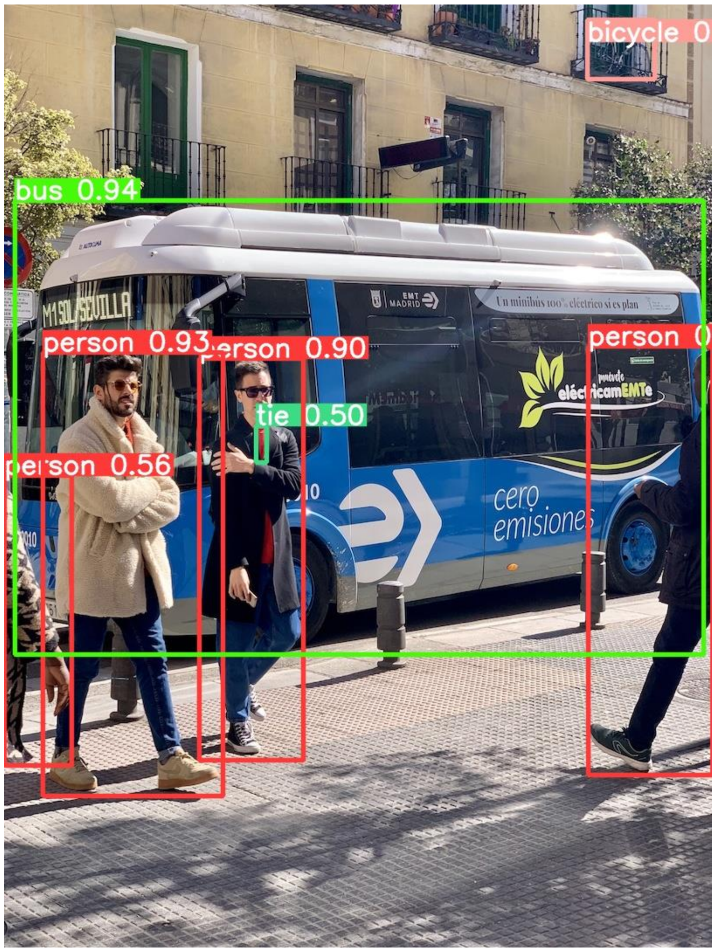
Code for addition of loss:



```
t = torch.full_like(ps[:, 5:], self.cn, device=device)
t[range(n), tcls[i]] = self.cp
# --- YOUR CODE GOES HERE ---
lcls += self.BCEcls(ps[:, 5:], t)
# --------------------------
```



Output: bus.jpg:



Output: cats_and_dogs.jpg (enlarged):