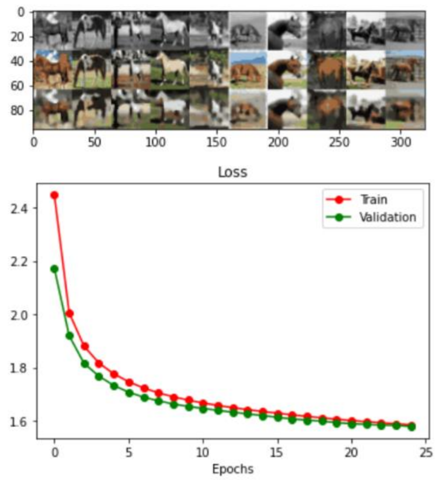
Part A code

```
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, stride=2),
            nn.BatchNorm2d(num_features=num_filters),
            nn.ReLU()
        )
        self.block2 = nn.Sequential(
            nn.Conv2d(num_filters, 2*num_filters, kernel, padding = padding),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.BatchNorm2d(num_features=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_features=num_filters),
            nn.ReLU()
        )
        self.block4 = nn.Sequential(
            nn.Conv2d(num_filters, num_colours, kernel, padding = padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_features=num_colours),
            nn.ReLU(),
            nn.Conv2d(num_colours, num_colours, kernel, padding=padding)
        )
```

```python
def forward(self, x):
    ############### YOUR CODE GOES HERE ###############
    h1 = self.block1(x)
    h2 = self.block2(h1)
    h3 = self.block3(h2)
    output = self.block4(h3)

    return output
    ###################################################
```

Part A visualization and comments:



The results are not good at all since the output graphs are quite vague and blurred, and the validation accuracy has only 41.2% after 25 epochs.

Part A Q3 six values to report:

To calculate the number of weights, outputs and connections, I follow the formula from the lecture slides, and here I would show several layers' computation methods, and the rest would be pretty much the same. From the images to the first convolutional layer, we have the output dimensions as (assuming kernel size k being odd):

$$\begin{aligned}\hat{W} = \hat{H} &= [(32 - k + 2 \times (k//2))/1 + 1] \\ &= 32 - k + 2 \times \frac{k-1}{2} + 1 \\ &= 32\end{aligned}$$

and so we have:

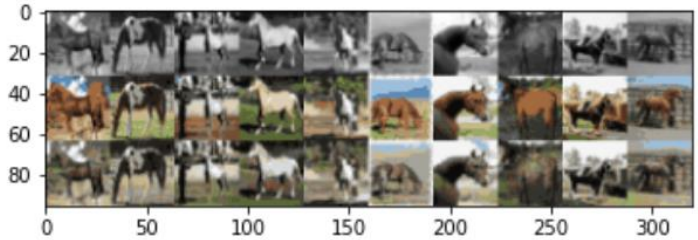
- Number of weights: $k^2 \times \text{NIC} \times \text{NF}$
- Number of outputs: $32^2 \times NF = 1024 \times NF$
- Number of connections: $32^2 \times k^2 \times NF \times NIC = 1024 \times k^2 \times NF \times NIC$ From the first convolutional layer to the first max-pooling layer, since the max-pooling does not have parameters, the number of weights would be 0, which is the same situation when dealing with the Upsampling layer. Since we are assuming the kernel size and stride of the max-pooling layer to be 2, we can calculate the number of outputs and connections similar as above. Following the above procedure, the final 6 figures to report are following:
- When each input dimension (width/height) is not doubled (original input):
  - Number of weights: $k^2 \times NIC \times NF + 4k^2 \times NF^2 + k^2 \times NC \times NF + K^2 \times NC^2$
  - Number of outputs: $2240 \times NF + 2304 \times NC$
  - Number of connections:
    $$1024k^2 \times NF \times NIC + 1792 \times NF + 640 \times K^2 \times NF^2 + 256k^2 \times NC \times NF + 1024 \times NC + 1024k^2 \times NC^2$$
- When each input dimension (width/height) is doubled, we would have the original answers multiplied by 4 since we have two dimensions for an image, and each dimension is being doubled:
  - Number of weights: $4 \times (k^2 \times NIC \times NF + 4k^2 \times NF^2 + k^2 \times NC \times NF + K^2 \times NC^2)$
  - Number of outputs: $4 \times (2240 \times NF + 2304 \times NC)$
  - Number of connections:
    $$4 \times (1024k^2 \times NF \times NIC + 1792 \times NF + 640 \times K^2 \times NF^2 + 256k^2 \times NC \times NF + 1024 \times NC + 1024k^2 \times NC^2)$$

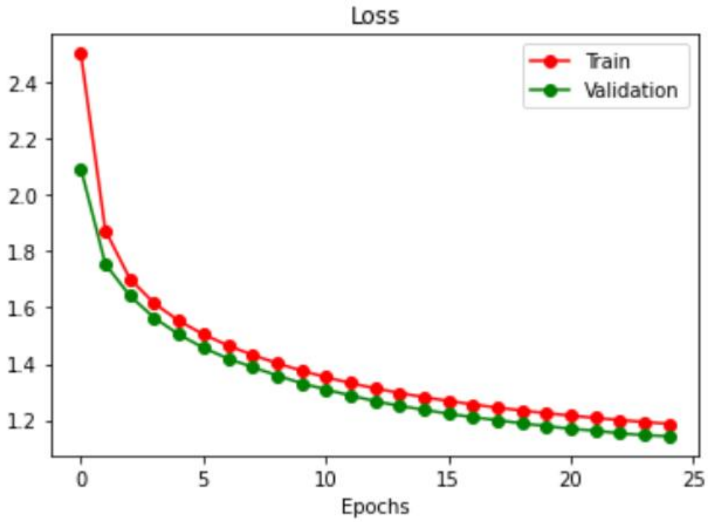
Part B Q1 code:

```
################ 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_features=num_filters),
    nn.ReLU()
)

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

self.block4 = nn.Sequential(
    nn.ConvTranspose2d(num_filters, num_colours, kernel_size=kernel,
                       stride = 2, padding = 1, output_padding = 1,
                       dilation=1),
    nn.BatchNorm2d(num_features=num_colours),
    nn.ReLU(),
    nn.Conv2d(num_colours, num_colours, kernel_size=kernel, padding = 1)
)
```

```
def forward(self, x):
    ################ YOUR CODE GOES HERE ################
    h1 = self.block1(x)
    h2 = self.block2(h1)
    h3 = self.block3(h2)
    output = self.block4(h3)

    return output
    #####################################################
```

Part B Q2 visualization:





The validation accuracy is better than the previous model.

Part B Q3-5:

▾ Questions 3 – 5

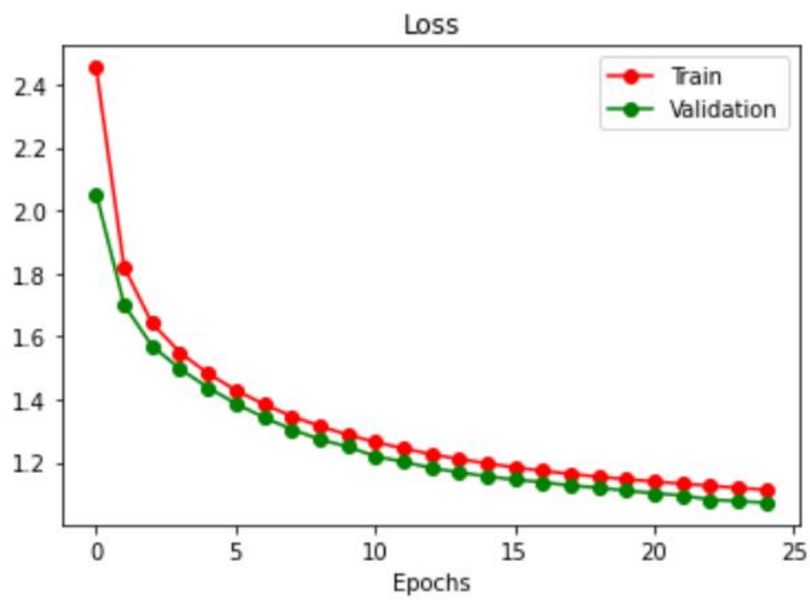
- Q3: The results are better than what in part A, as we can notice that the validation accuracy has come to 55.3% and the graphs are clearer with more accurate color pixels right now. The validation loss for the **ConvTransposeNet** is 1.1426, which is lower than 1.5788, which is the validation loss for the **PoolUpsampleNet**. The reason for this case might be that the *Upsample* layer has no trainable parameters, but *ConvTranspose2d* layer has trainable parameters, which can capture the gradient information from loss and so it can further update the model.
- Q4: To make sure the model is in the same shape after the first two $\mathrm{nn.Conv2d}$ layers, for kernel size being 4, we need $\mathrm{padding} = 1$, and for kernel size being 5, we need $\mathrm{padding} = 2$ since $\mathrm{torch.nn}$ calculate the output size using round-down operation in math. The formula for calculating the padding for kernel size being 5 would be:
$$\lfloor \frac{32 + 2 \times \mathrm{padding} - 5}{2} + 1 \rfloor = 16$$
and the formula to calculate the padding when kernel size is 4 is similar except replacing 5 into 4 in the above formula.
For the $nn.ConvTranspose2d$ layers, we need $2 \times \mathrm{padding} - \mathrm{output\_padding} = 2$ when kernel size is 4, and $2 \times \mathrm{padding} - \mathrm{output\_padding} = 3$ when kernel size is 5. The formula for kernel size being 5 would be:
$$(8 - 1) \times 2 - 2 \times \mathrm{padding} + 1 \times (5 - 1) + \mathrm{output\_padding} + 1 = 16$$
and it is similar when kernel size is 4 with replacing of 5 into 4 in the above formula.
- Q5: As the batch size increases, the training/validation loss increases as well, and the validation accuracy decreases, which means the quality of the output images are decreasing.

Part C Q1 code:

```
################ 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_features=num_filters),
    nn.ReLU()
)

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

self.block4 = nn.Sequential(
    nn.ConvTranspose2d(2*num_filters, num_colours, kernel_size=kernel,
                       stride = 2, padding = 1, output_padding = 1,
                       dilation=1),
    nn.BatchNorm2d(num_features=num_colours),
    nn.ReLU()
)
self.conv5 = nn.Conv2d(num_in_channels + num_colours, num_colours,
                       kernel_size=kernel, padding = 1)
```

```python
def forward(self, x):
    ############### YOUR CODE GOES HERE ###############
    h1 = self.block1(x)
    h2 = self.block2(h1)
    h3 = self.block3(h2)
    h3 = torch.cat((h3, h1), 1)
    h4 = self.block4(h3)
    h4 = torch.cat((h4, x), 1)

    output = self.conv5(h4)

    return output
    ###################################################
```

Part C Q2 curve:



Part C Q3:

Question 3

The result is better than the previous model qualitatively since now we can see the graphs much clearer with more accurate color pixels than the previous model, and the validation loss and accuracy are both improved by the skip connections. The reasons for skip connections improving our CNN models are:

1. The *skip connections* can provide the relevant locality information where it is needed by when doing the *Upsampling* or *ConvTranspose2d*, we find the last layer before the image is shrinked, where the image still had the same size and simply add it pixel-wise to the upsampled image so that the now upsampled feature map has both the locality information lost in shrinking the image, say max-pooling or strided convolution layers, but also the dominant info after shrinking the image. This allows for much better detail in the task like prediction color pixels where we need both local information around the predicted pixel and also the global information in the image with previous size.
2. The *skip connections* will have more trainable parameters so that we can capture more information from the loss in the backward propagation to update our model better.

Part D.1 code:

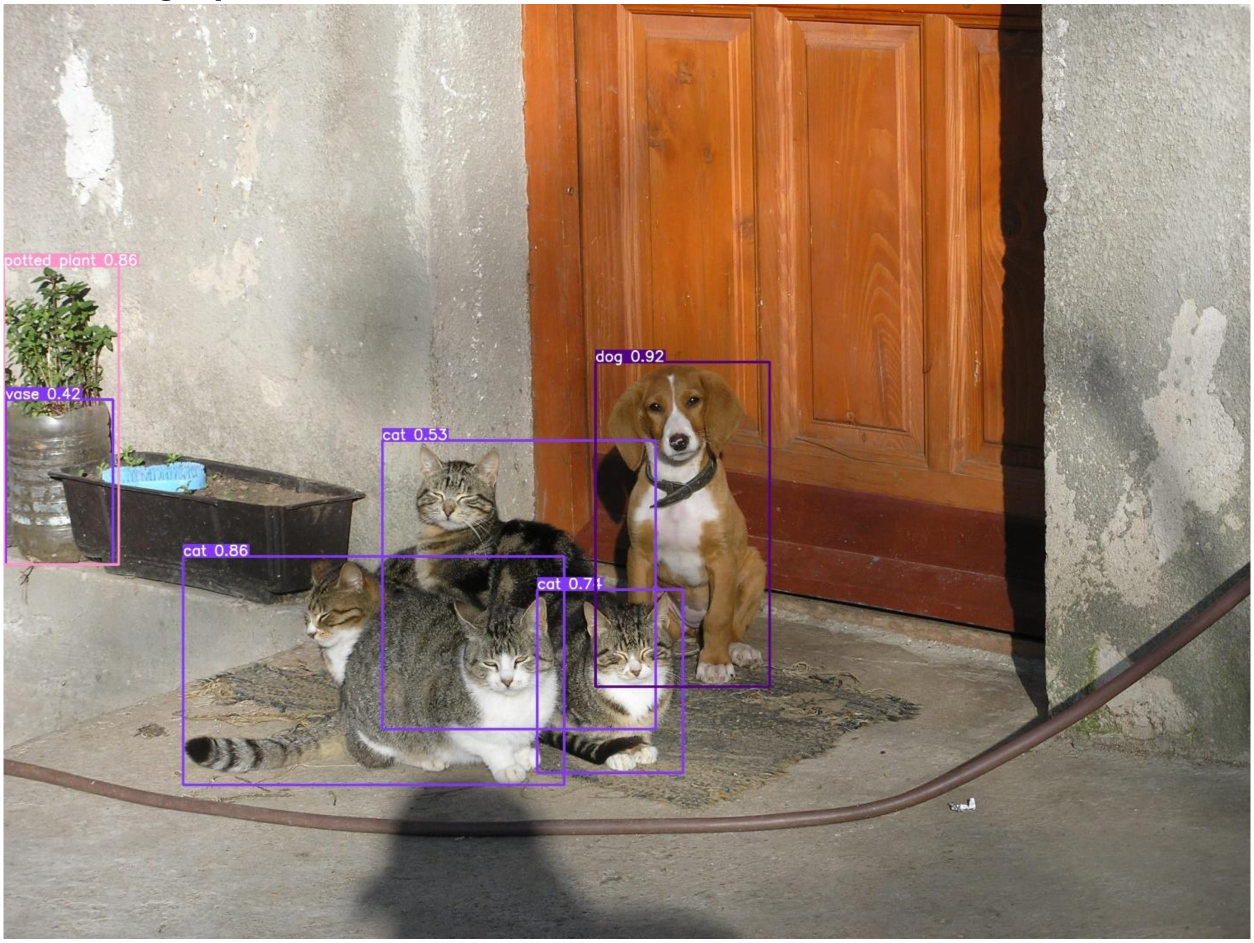
```
freeze = [f'model.{x}.' for x in range
for k, v in model.named_parameters():
        # --- YOUR CODE GOES HERE ---
        if any(x in k for x in freeze):
            v.requires_grad = False
        # ---------------------------
```

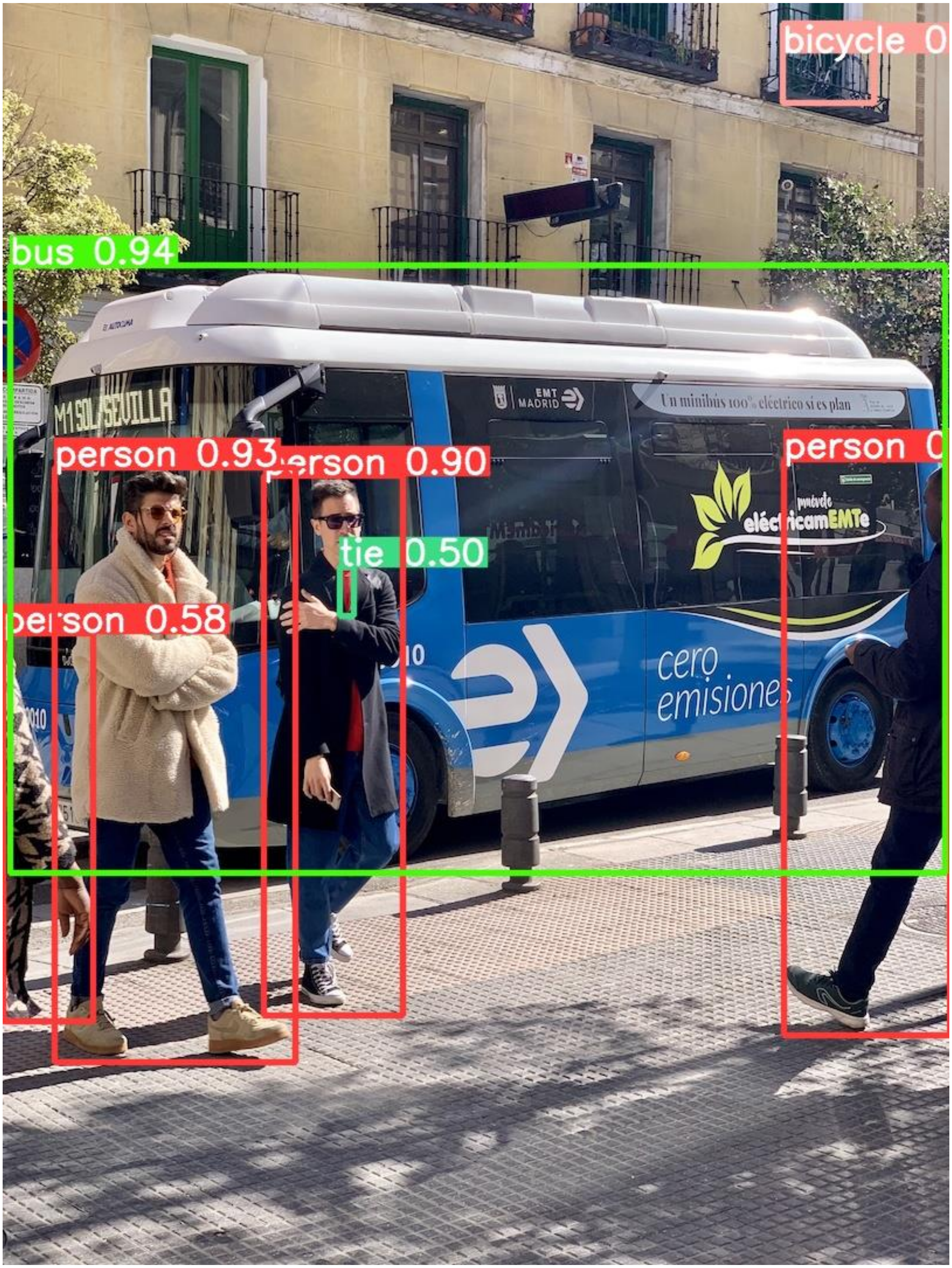
Part D.2 code:

```
# --- 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))
```

```
if self.nc > 1:  # cls loss (only if multiple classes)
        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)
        # ---------------------------
```

Part D.2 graphs:




bicycle 0
bus 0.94
M1 SOL/SEVILLA
EMT MADRID
Un minibus 100% eléctrico si es plan
person 0.93
person 0.90
person 0
tie 0.50
person 0.58
muévete
eléctricamEMTe
cero
emisiones