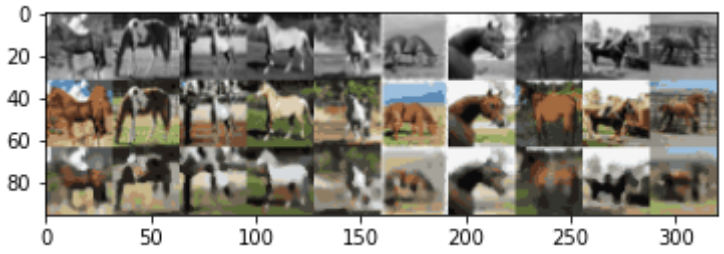
Programming Assignment 2

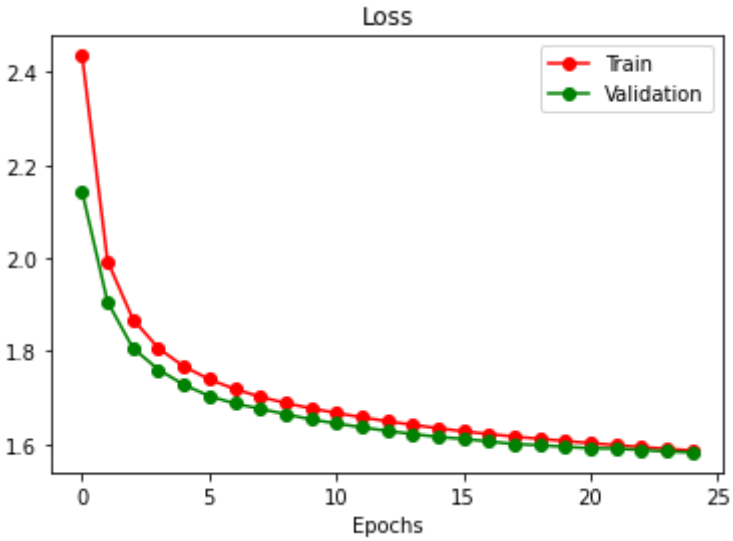
Part A: Pooling and Upsampling

A.1 Implementation of PoolUpsampleNet

```
[7]
    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.firstLayer = nn.Sequential(
                nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, padding=padding),
                nn.MaxPool2d(2),
                nn.BatchNorm2d(num_filters),
                nn.ReLU()
            )

            self.secondLayer = nn.Sequential(
                nn.Conv2d(num_filters, 2*num_filters, kernel_size=kernel, padding=padding),
                nn.MaxPool2d(2),
                nn.BatchNorm2d(2*num_filters),
                nn.ReLU()
            )

            self.thirdLayer = nn.Sequential(
                nn.Conv2d(2*num_filters, num_filters, kernel_size=kernel, padding=padding),
                nn.Upsample(scale_factor=2),
                nn.BatchNorm2d(num_filters),
                nn.ReLU()
            )

            self.fourthLayer = 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.lastLayer = nn.Conv2d(num_colours, num_colours, kernel, padding=padding)
            ###################################################

        def forward(self, x):
            ############### YOUR CODE GOES HERE ###############
            first = self.firstLayer(x)
            second = self.secondLayer(first)
            third = self.thirdLayer(second)
            fourth = self.fourthLayer(third)
            return self.lastLayer(fourth)
            ###################################################
```

A.2 Training Result





The shown figure is the result obtained from the main training loop of PoolUpsampleNet. It does not look good to me as the images are quite blurry with greyscale pixels present. Also, after 25 epoch, accuracy is around 41.4%, which is quite low.

A.3

Assume the kernel size is k

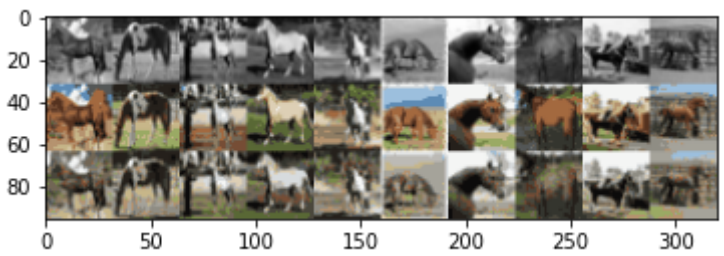
- when each input dimension (width/height) is not doubled (original input)
  - number of weights = $k^2(NIC \cdot NF + NF^2 + 2NF^2 + NF^2 + NC \cdot NF)$= $k^2(NIC \cdot NF + 4NF^2 + NC \cdot NF)$
  - number of outputs = $2880NF + 3328NC$
  - number of connections = $k^2(1024 \cdot NIC \cdot NF + 640NF^2 + 256NF \cdot NC)$
- when each input dimension (width/height) is doubled
  - number of weights = $k^2(NIC \cdot NF + NF^2 + 2NF^2 + NF^2 + NC \cdot NF)$= $k^2(NIC \cdot NF + 4NF^2 + NC \cdot NF)$
  - number of outputs = $11520NF + 13312NC$
  - number of connections = $4k^2(1024 \cdot NIC \cdot NF + 640NF^2 + 256NF \cdot NC)$

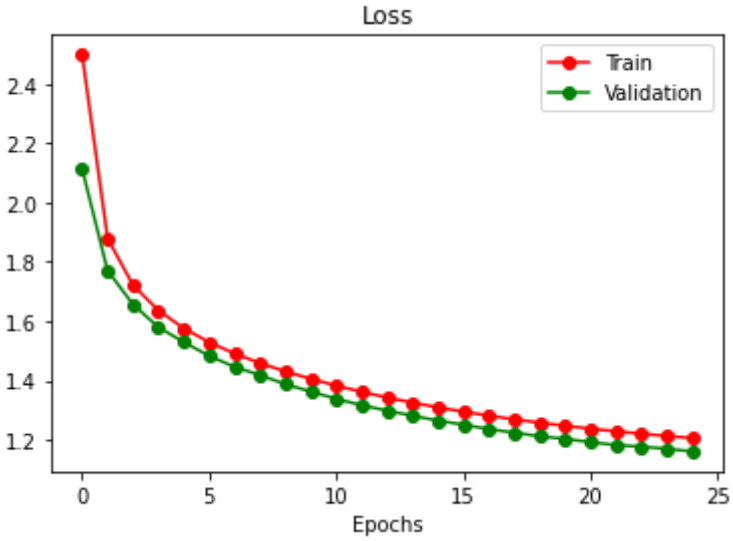
Part B: Strided and Transposed Dilated Convolutions

B.1 Implementation of ConvTransposeNet

```
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.firstLayer = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, padding=padding, stride=stride),
            nn.BatchNorm2d(num_filters),
            nn.ReLU()
        )

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

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

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

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

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        first = self.firstLayer(x)
        second = self.secondLayer(first)
        third = self.thirdLayer(second)
        fourth = self.fourthLayer(third)
        return self.lastLayer(fourth)
        ###################################################
```

B.2





B.3

The trained result from ConvTransposeNet model seems better from what we got from PoolUpsampleNet because

- the image seems less blurry
- the validation accuracy increases from 41.4% to 54.7%.
- the validation loss decreases from 1.5848 to 1.1565.

We know that Max-pooling can reduce the height and width of the output (i.e. reduce the input dimensionality) so that there will be less parameters to learn from for the next layer. Therefore, by removing MaxPool2d and Upsample from the model, ConvTransposeNet allows more features to be learnt within the system than what's of PoolUpsampleNet. Therefore, ConvTransposeNet yields a better result.

B.4

- if kernel size = 4,
  - padding parameter passed to the first two nn.Conv2d layers = 1
  - padding parameter passed to the nn.ConvTranspose2d layers= 1
  - output_padding parameter passed to the nn.ConvTranspose2d layers = 0
- if kernel size = 5,
  - padding parameter passed to the first two nn.Conv2d layers = 2
  - padding parameter passed to the nn.ConvTranspose2d layers = 2
  - output_padding parameter passed to the nn.ConvTranspose2d layers = 1

B.5

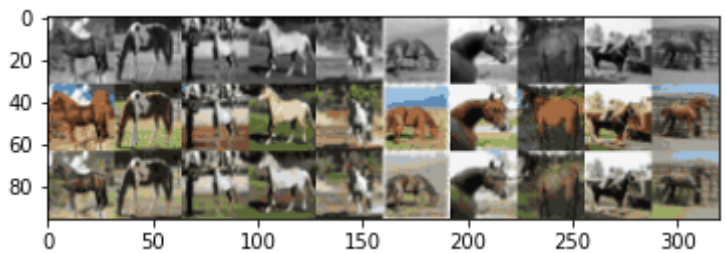
When fixing the number of epochs, increase in batch sizes will result in a increase in validation loss and a worse final image output quality. Vice versa.

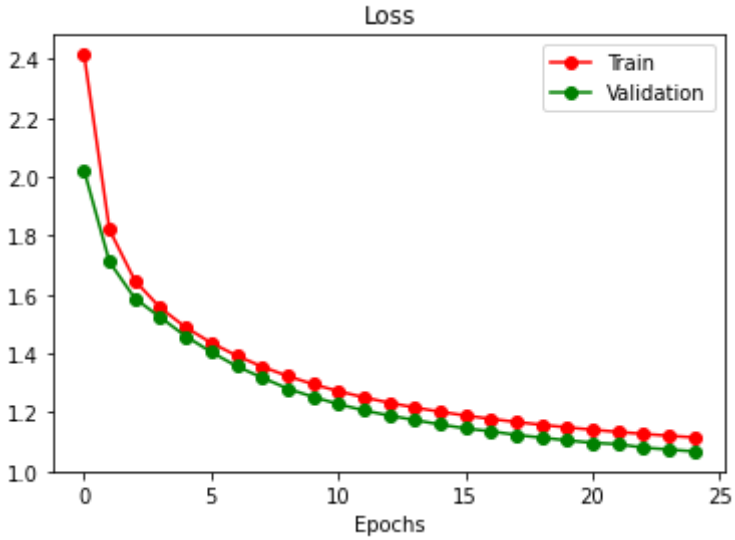
Part C: Skip Connections

C.1 Implementation of UNet

```
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.firstLayer = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, padding=padding, stride=stride),
            nn.BatchNorm2d(num_filters),
            nn.ReLU()
        )

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

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

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

        self.lastLayer = nn.Conv2d(num_in_channels + num_colours, num_colours, kernel_size=kernel,
                                   padding=padding)
        ###################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        first = self.firstLayer(x)
        second = self.secondLayer(first)
        third = self.thirdLayer(second)
        fourth = self.fourthLayer(torch.cat([third, first], 1))
        return self.lastLayer(torch.cat([fourth, x], 1))
        ###################################################
```

C.2





C.3

First of all, comparing the result obatined from UNet to the results from Part A and Part B, we see that:

- the image obtained from skip connections seems to be the least blurry most among all three
- the validation loss from UNet is 1.0512 after 25 epoch, which is smaller than 1.5848 (from Part A) and 1.1565 (from Part B).
- the validation accuracy is 58.9% which is greater than 41.4% (from Part A) and 54.7% (from Part B).

Therefore, we can conclude that skip connections does improve the validation loss and accuracy.

Two reasons why skip connections might improve the performance of our CNN models are:

- There are more learning parameters in skip connection models, which results in a better performance. For example, output of first layer and third layer are both the source of input for the fourth layer. But for the previous models, they only have output from the third layer as the source for the fourth layer.
- Skip connection concatenates the layers. By doing so, the features can be resused multiple time and thus a better result.

Part D: Object Detection

D.1 Fine-tuning from pre-trained models for object detection

```
# Freeze
freeze = [f'model.{x}.' for x in range(freeze)]  # layers to fr
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
    # ---------------------------
```

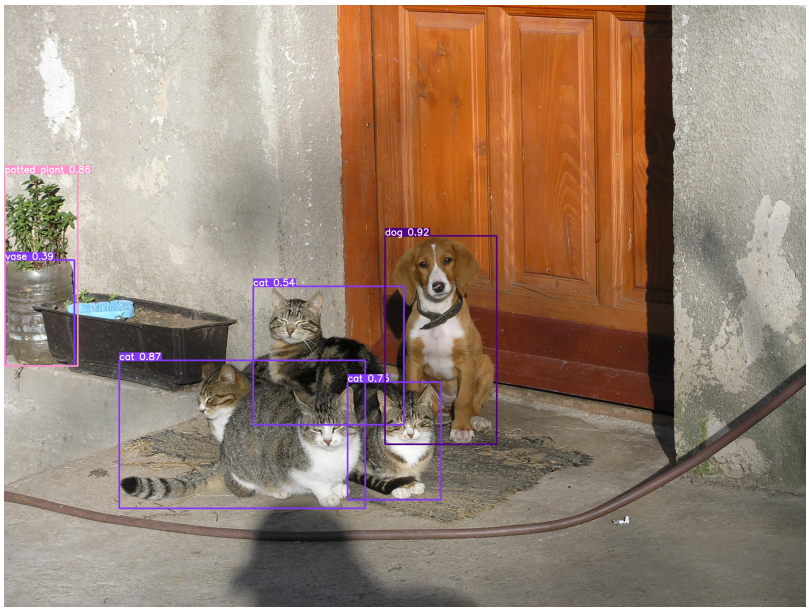
D.2 Implement the classification loss

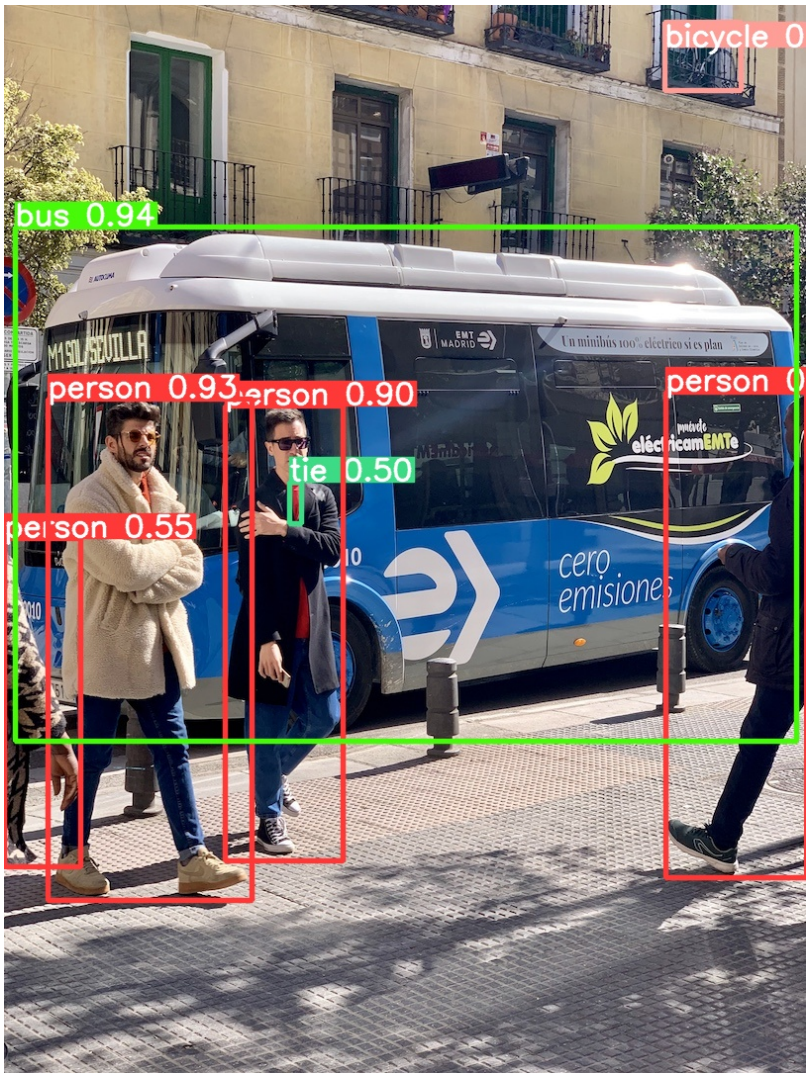
D.2.1

```
95          # Define loss criteria
96          # --- YOUR CODE GOES HERE ---
97          BCEcls = nn.BCEWithLogitsLoss(pos_weight=torch.tensor([h['cls_pw']], device=device))
98          # ---------------------------
99          BCEobj = nn.BCEWithLogitsLoss(pos_weight=torch.tensor([h['obj_pw']], device=device))
100
101         # Class label smoothing https://arxiv.org/pdf/1902.04103.pdf eqn 3
102         self.cp, self.cn = smooth_BCE(eps=h.get('label_smoothing', 0.0))  # positive, negative BCE ta
103
```

```
145                 # Classification
146                 if self.nc > 1:  # cls loss (only if multiple classes)
147                     t = torch.full_like(ps[:, 5:], self.cn, device=device)  # targets
148                     t[range(n), tcls[i]] = self.cp
149                     # --- YOUR CODE GOES HERE ---
150                     lcls += self.BCEcls(ps[:,5:],t)
151                     # ---------------------------
152
```

D.2.2




bicycle 0
bus 0.94
person 0.93
person 0.90
person 0
tie 0.50
person 0.55
cero emisiones