CSC413PA2

Che Liu, 1002246839

February 27, 2020

Part A: Colourization as Classification (2 pts)

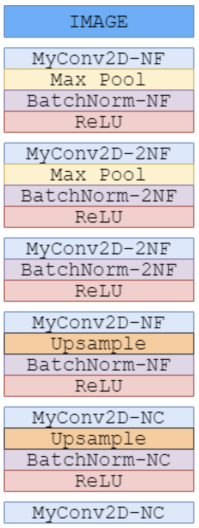

Figure 1: Network architecture

1. Complete the model CNN.

```
class CNN(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super(CNN, self).__init__()
        padding = kernel // 2

        ############### YOUR CODE GOES HERE ###############
        self.conv1 = nn.Sequential(
            MyConv2d(num_in_channels, num_filters, kernel, padding=padding),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(inplace=True))
        self.conv2 = nn.Sequential(
            MyConv2d(num_filters, 2*num_filters, kernel, padding=padding),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(2*num_filters),
            nn.ReLU(inplace=True))
        self.conv3 = nn.Sequential(
            MyConv2d(2*num_filters, 2*num_filters, kernel, padding=padding),
            nn.BatchNorm2d(2*num_filters),
            nn.ReLU(inplace=True))
        self.conv4 = nn.Sequential(
            MyConv2d(2*num_filters, num_filters, kernel, padding=padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(inplace=True))
        self.conv5 = nn.Sequential(
            MyConv2d(num_filters, num_colours, kernel, padding=padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_colours),
            nn.ReLU(inplace=True))
        self.conv6 = nn.Sequential(
            MyConv2d(num_colours, num_colours, kernel, padding=padding))
        ###################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = self.conv6(x)

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

Figure 2: Code for network definition and training

2. Run main training loop of CNN. 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?

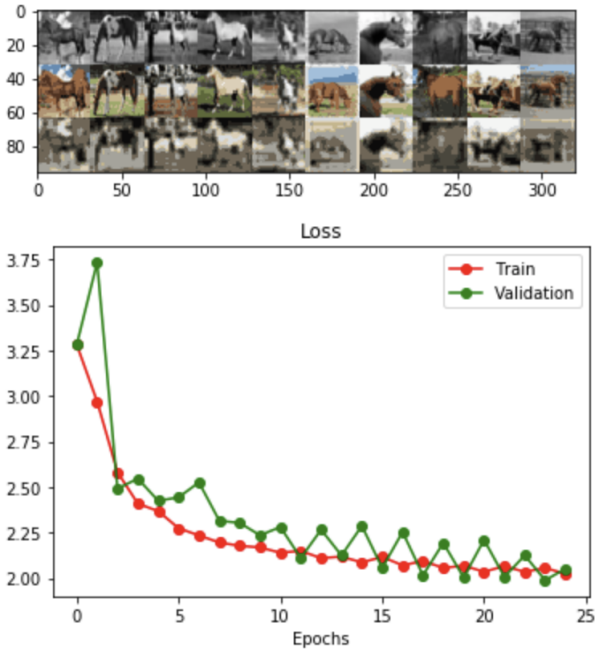

Figure 3: Result and training error with 25 epochs

The trained result doesn't look good. The output images barely have any color in them; they look vastly different from the ground truth images.

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

Assumptions:
For the number of weights, convolution layers and batch norm layers are considered.
For the number of outputs, the convolution layers and pooling layers are considered.
For the number of connections, only the convolution layers are considered.

let k denote the kernel size.
let n1 denote the number of input channels in the first layer.

With default input dimension:
# of weights: (NF*(k*k*n1+3))+(2NF*(k*k*NF+3))+(2NF*(k*k*2NF+3))+(NF*(k*k*2NF+3))+(NC*(k*k*NF+3))+(NC*(k*k*NC+1))
# of outputs:(NF*(32*32+16*16))+(2NF*(16*16+8*8))+(2NF*8*8)+(NF*(8*8+16*16))+(NC*(16*16+32*32))+(NC*32*32)
# of connections:(NF*32*32*(k*k*n1+1))+(2NF*16*16*(k*k*NF+1))+(2NF*8*8*(k*k*2NF+1))+(NF*8*8*(k*k*2NF+4))+(NC*16*16*(k*k*NF+4))+(NC*32*32*(k*k*NC+1))

After doubling input dimension:
# of weights: (NF*(k*k*n1+3))+(2NF*(k*k*NF+3))+(2NF*(k*k*2NF+3))+(NF*(k*k*2NF+3))+(NC*(k*k*NF+3))+(NC*(k*k*NC+1))
# of outputs:(NF*(64*64+32*32))+(2NF*(32*32+16*16))+(2NF*16*16)+(NF*(16*16+32*32))+(NC*(32*32+64*64))+(NC*64*64)
# of connections:(NF*64*64*(k*k*n1+1))+(2NF*32*32*(k*k*NF+1))+(2NF*16*16*(k*k*2NF+1))+(NF*16*16*(k*k*2NF+4))+(NC*32*32*(k*k*NF+4))+(NC*64*64*(k*k*NC+1))

4. Consider an pre-processing step where each input pixel is multiplied elementwise by scalar a, and is shifted by some scalar b. That is, where the original pixel value is denoted x, the new value is calculated y = ax + b. Assume this operation does not result in any overflows. How does this pre-processing step affect the output of the conv net from Question 1 and 2?

Batch normalization makes the output invariant to the scale and shift of the previous activations. Therefore, the scaling and shift pre-processing has no effect on the conv net output.

Part B: Skip Connections (2 pts)

1. Complete the model UNet.

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

        ############## YOUR CODE GOES HERE ##############
        # x to conv6
        # conv1 to conv5
        padding = kernel // 2
        self.conv1 = nn.Sequential(
            MyConv2d(num_in_channels, num_filters, kernel, padding=padding),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(inplace=True))
        self.conv2 = nn.Sequential(
            MyConv2d(num_filters, 2*num_filters, kernel, padding=padding),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(2*num_filters),
            nn.ReLU(inplace=True))
        self.conv3 = nn.Sequential(
            MyConv2d(2*num_filters, 2*num_filters, kernel, padding=padding),
            nn.BatchNorm2d(2*num_filters),
            nn.ReLU(inplace=True))
        self.conv4 = nn.Sequential(
            MyConv2d(2*num_filters, num_filters, kernel, padding=padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(inplace=True))
        self.conv5 = nn.Sequential(
            MyConv2d(num_filters+num_filters, num_colours, kernel, padding=padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_colours),
            nn.ReLU(inplace=True))
        self.conv6 = nn.Sequential(
            MyConv2d(num_colours+num_in_channels, num_colours, kernel, padding=padding))
        ################################################

    def forward(self, x):
        ############## YOUR CODE GOES HERE ##############
        # conv1 to conv6
        # conv2 to conv5
        x_conv1 = self.conv1(x)
        x_conv2 = self.conv2(x_conv1)
        x_conv3 = self.conv3(x_conv2)
        x_conv4 = self.conv4(x_conv3)
        x_conv5 = self.conv5(torch.cat((x_conv4,x_conv1),1))
        x_conv6 = self.conv6(torch.cat((x_conv5,x),1))

        return x_conv6
        ################################################
```

Figure 4: Code for UNet definition and training

2. Train the model for at least 25 epochs and plot the training curve using a batch size of 100.

3. How does the result compare to the previous model? Did skip connections improve the validation loss and accuracy? Did the skip connections improve the output qualitatively? How? Give at least two reasons why skip connections might improve the performance of our CNN models.

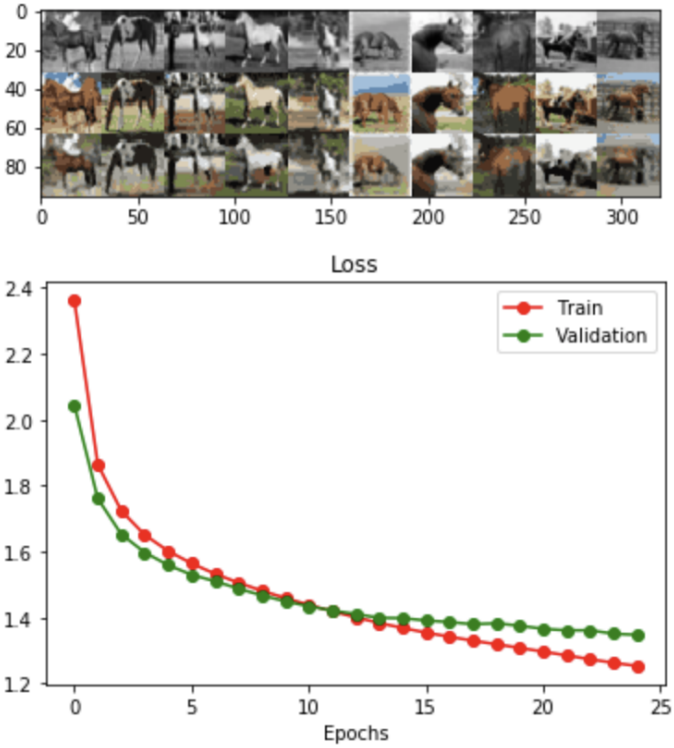

Figure 5: Result and training error with 25 epochs

The result looks better than the previous model. Skip connections improved the output qualitatively. Looking at the trained result, the model seems to learn that the horse should be brown and the grass should be green. Furthermore, skip connections also improved the validation loss and accuracy. The training curve look smoother than that of the previous model and the final validation error was lower than before. Skip connections might improve the performance of our CNN models because firstly it allows features learned in the earlier layers to directly flow into layers deeper in the network rather than potentially getting lost due to operations such as pooling. Furthermore, it helps with the vanishing gradient problem and makes training easier.

4. Re-train a few more "UNet" models using different mini batch sizes with a fixed number of epochs. Describe the effect of batch sizes on the training/validation loss, and the final image output.

The training curves and training results obtained using batch sizes of 1, 5, 10, 500, and 1000 are shown below:

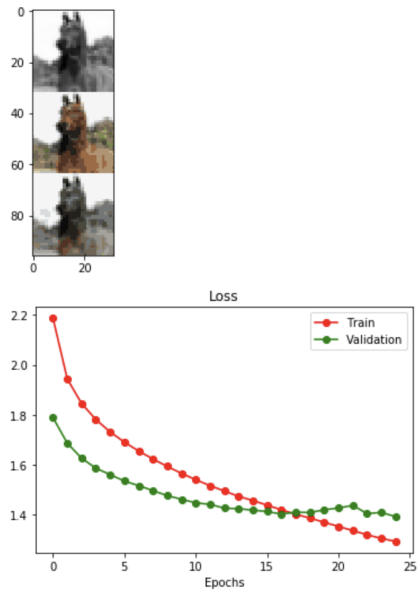

Figure 6: Result and training error with a batch size of 1

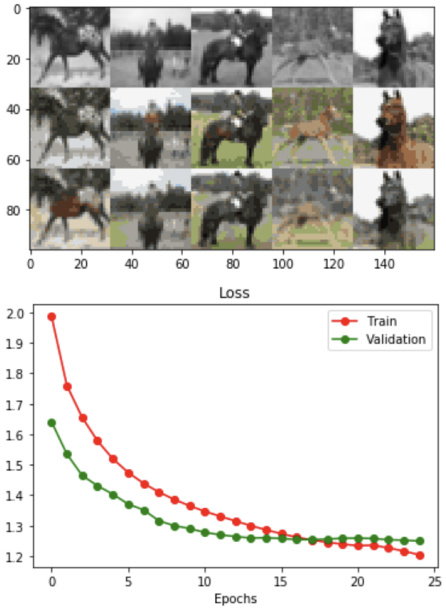

Figure 7: Result and training error with a batch size of 5

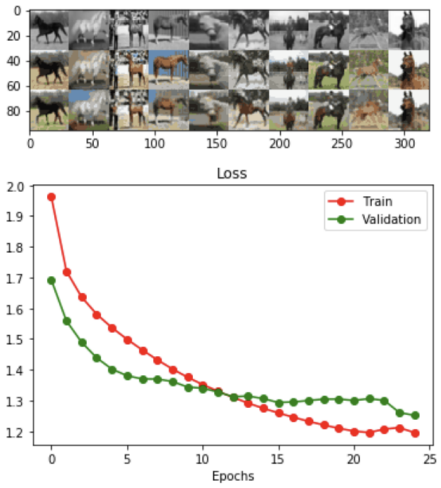

Figure 8: Result and training error with a batch size of 10

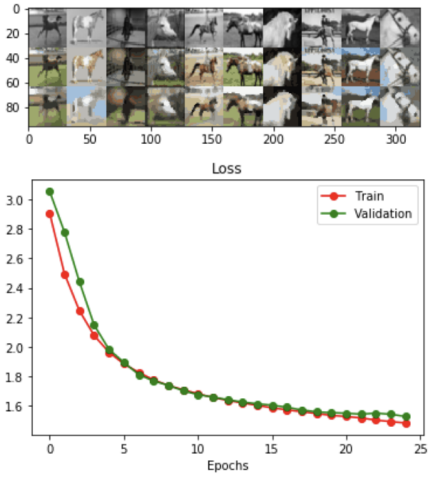

Figure 9: Result and training error with a batch size of 500

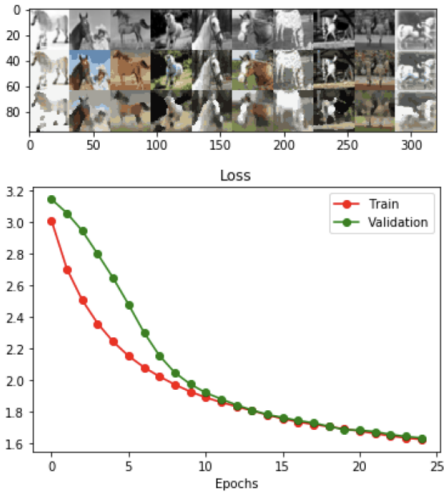

Figure 10: Result and training error with a batch size of 1000

The final validation accuracy seems to decrease with the batch size, with a batch size of 5 achieving the highest accuracy. The resulting images also look more and more natural when the batch size is getting smaller and smaller. The difference between training and validation error seems to be smaller when the batch size gets bigger. Batch size of 1 is an exception, potentially caused by the increased noise in gradient preventing the model from learning enough in only 25 epochs.

Part C: Fine-tune Semantic Segmentation Model (2 pts)

1. Complete the 'train' function in Part C of the notebook by adding 2-3 lines of code where indicated

```
# We only learn the last layer and freeze all the other weights
################ Code goes here ######################
# Around 2-3 lines of code
learned_parameters.append(model.classifier[4].weight)

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

Figure 11: Code to complete the 'train' function in Part C

2. Complete the script below by adding around 2 lines of code and train the model.

```
################ Code goes here ######################
# Around 2 lines of code
for param in model.parameters():
    param.requires_grad = False
model.classifier[-1] = nn.Conv2d(256, 2, 3)
model.classifier[-1].weight.requires_grad=True
#####################################################
```

Figure 12: Code to complete the 'AttrDict' class in Part C

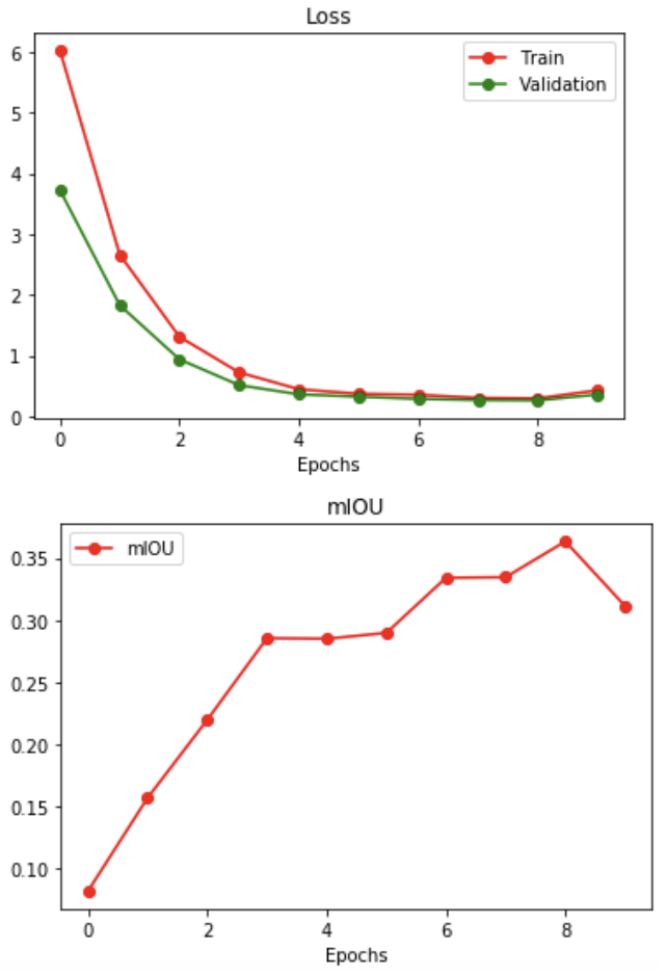

Figure 13: Loss and mIOU of the transfer learning model

3. Visualize predictions.

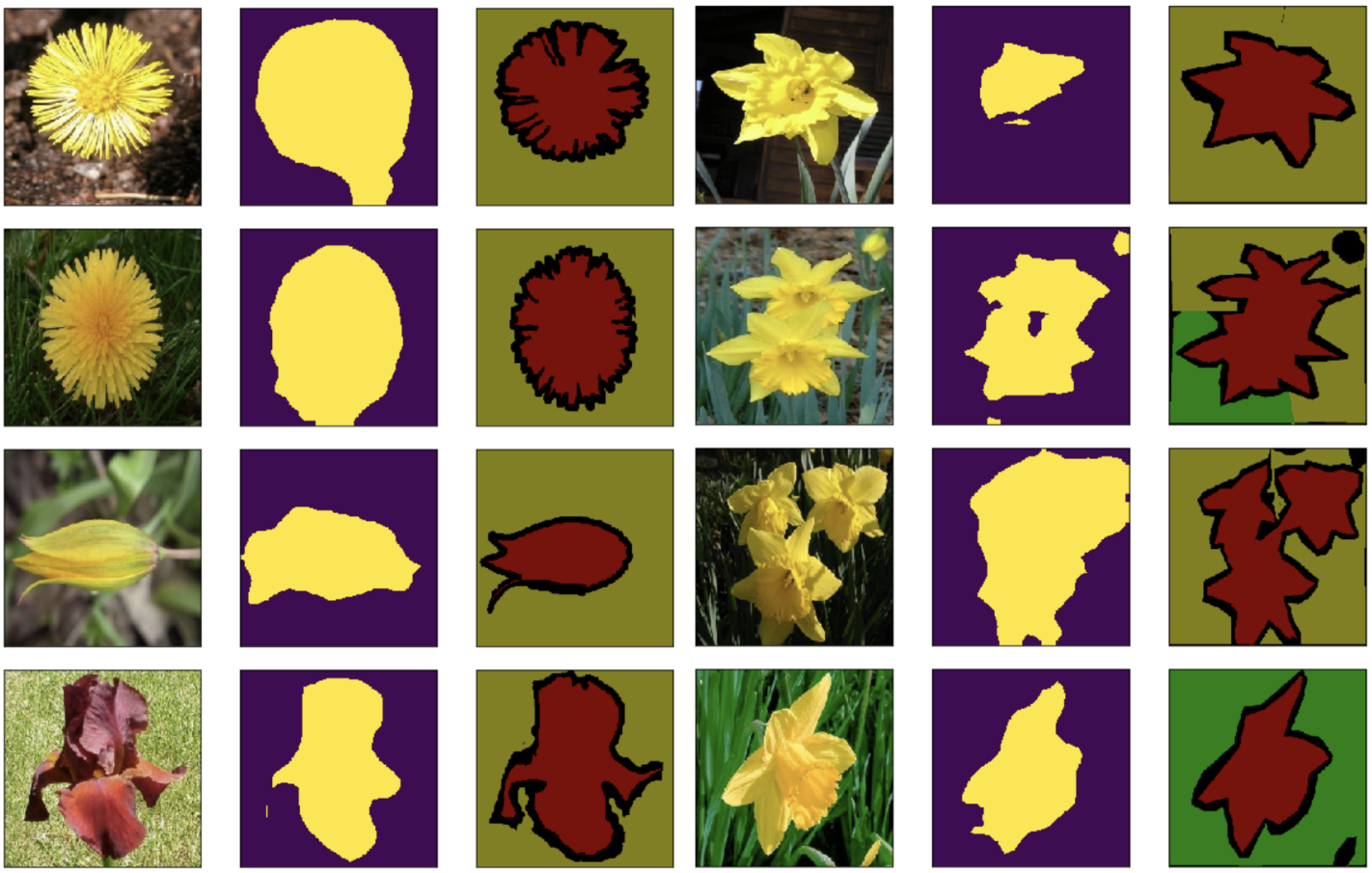

Figure 14: Visualization of the segmentation model

Figure 15: Visualization of the segmentation model

4. Consider a case of fine-tuning a pre-trained model with n number of layers. Each layers have the similar number of parameters, so the total number of parameters for the model is proportional to n. Describe the difference in memory complexity in terms of n between fine-tuning an entire pre-trained model versus fine-tuning only the last layer (freezing all the other layers). What about the computational complexity?

Suppose there are n layers in the pre-trained model and each layer consume similar memory and computation. Let m denote the memory required for fine-tuning each layer and c denote the computation required for fine-tuning each layer. Then:
Fine-tuning an entire pre-trained model has a memory complexity of O(mn) since all activations in all layers need to be stored in memory, whereas fine-tuning only the last layer has a memory complexity of O(m).
Fine-tuning an entire pre-trained model has a computation complexity of O(cn), whereas fine-truning only the last layer has a computation complexity of O(c).

5. If we increase the height and the width of the input image by a factor of 2, how does this affect the memory complexity of fine-tuning? What about the number of parameters?

Increasing the height and the width of the input image quadruples the memory complexity since in a segmentation problem we need to classify every pixel, and now we have four times as many pixels that need to be classified so the number of activations that need to be saved quadruples.. However, the number of parameters will not change because the number and dimension of the filters do not change.