CSC413 Programming Assignment 2: Convolutional Neural Networks

Tianyu Du (1003801647)

Wednesday 26th February, 2020

1 Part A: Colourization as Classification

Question 1 Implementation of the 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 ###############
        # Read hyper-parameters.
        self.padding = padding
        self.kernel = kernel
        self.num_filters = num_filters
        self.num_colours = num_colours
        self.num_in_channels = num_in_channels
        # Construct layers.
        # Block 1.
        self.block1 = nn.ModuleList([
            MyConv2d(
                in_channels=self.num_in_channels,
                out_channels=self.num_filters,
                kernel_size=self.kernel,
                padding=self.padding),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_features=self.num_filters),
            nn.ReLU()
        ])
        # Block 2.
        self.block2 = nn.ModuleList([
            MyConv2d(
                in_channels=self.num_filters,
                out_channels=2*self.num_filters,
                kernel_size=self.kernel,
```

```
                padding=self.padding),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_features=2*self.num_filters),
            nn.ReLU()
        ])
        # Block 3.
        self.block3 = nn.ModuleList([
            MyConv2d(
                in_channels=2*self.num_filters,
                out_channels=2*self.num_filters,
                kernel_size=self.kernel,
                padding=self.padding),
            nn.BatchNorm2d(num_features=2*self.num_filters),
            nn.ReLU()
        ])
        # Block 4.
        self.block4 = nn.ModuleList([
            MyConv2d(
                in_channels=2*self.num_filters,
                out_channels=self.num_filters,
                kernel_size=self.kernel,
                padding=self.padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_features=self.num_filters),
            nn.ReLU()
        ])
        # Block 5.
        self.block5 = nn.ModuleList([
            MyConv2d(
                in_channels=self.num_filters,
                out_channels=self.num_colours,
                kernel_size=self.kernel,
                padding=self.padding),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_features=self.num_colours),
            nn.ReLU()
        ])
        # last conv
        self.LastConv = MyConv2d(
            in_channels=self.num_colours,
            out_channels=self.num_colours,
            kernel_size=self.kernel,
            padding=self.padding
        )
        ####################################################

    def forward(self, x):
```

```
        ############### YOUR CODE GOES HERE ###############
        block_list = [
            self.block1, self.block2,
            self.block3, self.block4, self.block5]
        for block in block_list:
            for layer in block:
                x = layer(x)
        x = self.LastConv(x)
        return x
        ####################################################

```

Question 2 The result was terrible, the model merely add some brown pixels to the dark portion of the input image. Backgrounds such as sky and grassland are not properly identified and coloured.

Question 3 The table below summarizes configurations for all layers. For the ℓ^{th} layer. The number of weights reported includes biases as well, so that we are reporting number of (trainable) parameters. For example, the batch normalization layer has untrainable parameters (batch mean and batch standard deviation), but they are not reported here. We also assume that all convolution layers have the same kernel size k. The prefix of `B1.MyConv2D-NF` denotes the convolution layer in the first block of the entire network. There are

$$k^2\mathrm{NC}^2 + k^2\mathrm{NC} \times \mathrm{NF} + 8k^2\mathrm{NF}^2 + k^2\mathrm{NF} + 4\mathrm{NC} + 18\mathrm{NF} \tag{1.1}$$

$$= k^2\left(\mathrm{NC}^2 + \mathrm{NCNF} + 8\mathrm{NF}^2 + \mathrm{NF}\right) + 4\mathrm{NC} + 18\mathrm{NF} \tag{1.2}$$

weights (trainable parameters) in total. In particular, when $k = 3$,

$$9\mathrm{NC}^2 + 9\mathrm{NCNF} + 4\mathrm{NC} + 72\mathrm{NF}^2 + 27\mathrm{NF} \tag{1.3}$$

and for connections in convolution layers:

$$k^2\left(1024\mathrm{NC}^2 + 256\mathrm{NCNF} + 896\mathrm{NF}^2 + 1024\mathrm{NF}\right) \tag{1.4}$$

In particular, when $k = 3$,

$$9216\mathrm{NC}^2 + 2304\mathrm{NCNF} + 8064\mathrm{NF}^2 + 9216\mathrm{NF} \tag{1.5}$$

Table 1: Summary of ConvNet (32 ×32)

Layer	#Weights(Params)	Output Shape	#Connections
B1.MyConv2D-NF	k^2×NF + NF	32×32×NF	32×32×NF×k^2
B1.MaxPool	0	16×16×NF	
B1.BatchNorm-NF	2×NF	16×16×NF	
B1.ReLu	0	16×16×NF	
B2.MyConv2D-2NF	k^2×NF ×2NF + 2 ×NF	16×16×2 NF	16^2×NF ×2NF×k^2
B2.MaxPool	0	8×8×2×NF	
B2.BatchNorm-2NF	4×NF	8×8×2×NF	
B2.ReLu	0	8×8×2×NF	
B3.MyConv2D-2NF	k^2×2NF ×2NF + 2NF	8×8×2×NF	8^2×2NF ×2NF×k^2
B3.BatchNorm-2NF	4×NF	8×8×2×NF	
B3.ReLU	0	8×8×2×NF	
B4.MyConv2D-NF	k^2 ×2NF×NF + NF	8×8×NF	8^2 ×2NF ×NF ×k^2
B4.Upsample	0	16×16×NF	
B4.BatchNorm-NF	2×NF	16×16×NF	
B4.ReLU	0	16×16×NF	
B5.MyConv2D-NC	k^2×NF×NC + NC	16×16×NC	16^2×NF×NC×k^2
B5.Upsample	0	32×32×NC	
B5.BatchNorm-NC	2×NC	32×32×NC	
B5.ReLu	0	32×32×NC	
Out.MyConv2D-NC	k^2 ×NC ×NC + NC	32×32×NC	32^2×NC^2 ×k^2

When the image size is doubled, the total number of weights is unchanged. The weight and height of each output (i.e., the first two dimensions of the output) are doubled. The total number of connections is 400% of the network taking 32×32 image, the details are summarized in the following table:

Table 2: Summary of ConvNet (64 ×64)

Layer	#Weights(Params)	Output Shape	#Connections
B1.MyConv2D-NF	k^2×NF + NF	64×64×NF	64×64×NF×k^2
B1.MaxPool	0	32×32×NF	
B1.BatchNorm-NF	2×NF	32×32×NF	
B1.ReLu	0	32×32×NF	
B2.MyConv2D-2NF	k^2×NF ×2NF + 2 ×NF	32×32×2 NF	32^2×NF ×2NF×k^2
B2.MaxPool	0	16×16×2×NF	
B2.BatchNorm-2NF	4×NF	16×16×2×NF	
B2.ReLu	0	16×16×2×NF	
B3.MyConv2D-2NF	k^2×2NF ×2NF + 2NF	16×16×2×NF	16^2×2NF ×2NF×k^2
B3.BatchNorm-2NF	4×NF	16×16×2×NF	
B3.ReLU	0	16×16×2×NF	
B4.MyConv2D-NF	k^2 ×2NF×NF + NF	16×16×NF	16^2 ×2NF ×NF ×k^2
B4.Upsample	0	32×32×NF	
B4.BatchNorm-NF	2×NF	32×32×NF	
B4.ReLU	0	32×32×NF	
B5.MyConv2D-NC	k^2×NF×NC + NC	32×32×NC	32^2×NF×NC×k^2
B5.Upsample	0	64×64×NC	
B5.BatchNorm-NC	2×NC	64×64×NC	
B5.ReLu	0	64×64×NC	
Out.MyConv2D-NC	k^2 ×NC ×NC + NC	64×64×NC	64^2×NC^2 ×k^2

The number of trainable parameters are

$$k^2\left(\mathrm{NC}^2 + \mathrm{NCNF} + 8\mathrm{NF}^2 + \mathrm{NF}\right) + 4\mathrm{NC} + 18\mathrm{NF} \tag{1.6}$$

The number of connections in convolution layers is

$$4k^2\left(1024\mathrm{NC}^2 + 256\mathrm{NCNF} + 896\mathrm{NF}^2 + 1024\mathrm{NF}\right) \tag{1.7}$$

Question 4 The colourization should be exactly the same. The proposed affine preprocessing is equivalently adding one more linear layer before the first convolution layer. Since convolution operator is linear, the composite of the additional linear layer and the convolution layer is linear as well. The optimizer should be able to adjust weights in the convolution layer to offset the linear layer. In particular, each weight $\mathbf{w}$ will be adjusted to $\frac{\mathbf{w}-b}{a}$.

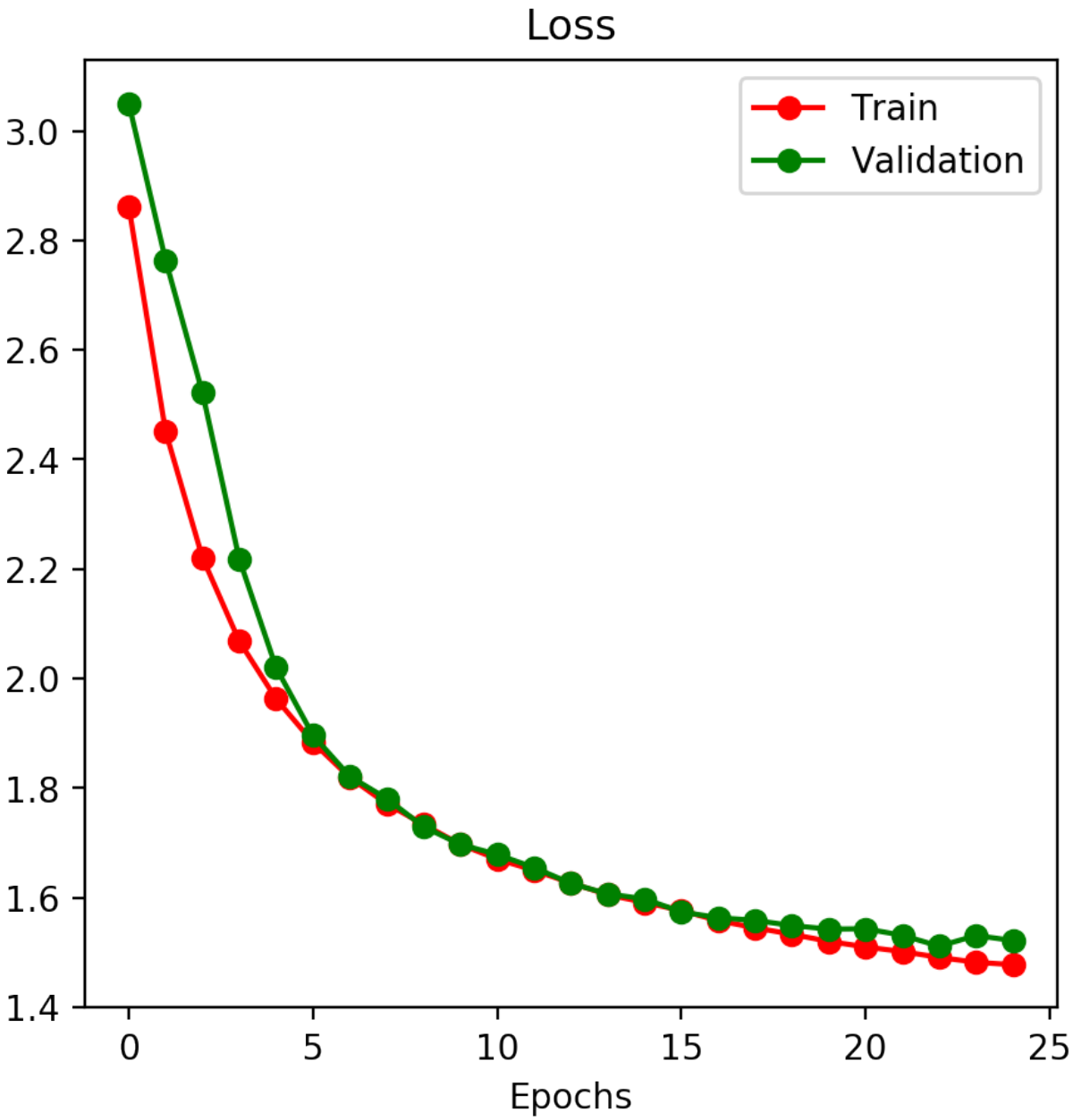
2 Part B: Skip Connections

Question 1 Implementation CNN with skip connections.

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

        ############### YOUR CODE GOES HERE ###############
        # Read hyper-parameters.
        self.kernel = kernel
        self.num_filters = num_filters
        self.num_colours = num_colours
        self.num_in_channels = num_in_channels
        # Construct layers.
        # Block 1.
        self.block1 = nn.Sequential(
            MyConv2d(
                in_channels=self.num_in_channels,
                out_channels=self.num_filters,
                kernel_size=self.kernel),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_features=self.num_filters),
            nn.ReLU()
        )
        # Block 2.
        self.block2 = nn.Sequential(
            MyConv2d(
                in_channels=self.num_filters,
                out_channels=2*self.num_filters,
                kernel_size=self.kernel),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_features=2*self.num_filters),
            nn.ReLU()
        )
        # Block 3.
        self.block3 = nn.Sequential(
            MyConv2d(
                in_channels=2*self.num_filters,
                out_channels=2*self.num_filters,
                kernel_size=self.kernel),
            nn.BatchNorm2d(num_features=2*self.num_filters),
            nn.ReLU()
        )
        # ===================================================
        # Block 4.
        self.block4 = nn.Sequential(
            MyConv2d(
                in_channels=4*self.num_filters,
                out_channels=self.num_filters,
                kernel_size=self.kernel),
```

```
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_features=self.num_filters),
            nn.ReLU()
        )
        # Block 5.
        self.block5 = nn.Sequential(
            MyConv2d(
                in_channels=2*self.num_filters,
                out_channels=self.num_colours,
                kernel_size=self.kernel),
            nn.Upsample(scale_factor=2),
            nn.BatchNorm2d(num_features=self.num_colours),
            nn.ReLU()
        )
        # last conv
        self.LastConv = MyConv2d(
            in_channels=self.num_colours + self.num_in_channels,
            out_channels=self.num_colours,
            kernel_size=self.kernel
        )
        ########################################################

    def forward(self, x):
        ############### YOUR CODE GOES HERE ###############
        block_1_out = self.block1(x)
        block_2_out = self.block2(block_1_out)
        block_3_out = self.block3(block_2_out)

        block_4_in = torch.cat((block_2_out, block_3_out), dim=1)
        block_4_out = self.block4(block_4_in)

        block_5_in = torch.cat((block_1_out, block_4_out), dim=1)
        block_5_out = self.block5(block_5_in)

        last_conv_in = torch.cat((x, block_5_out), dim=1)
        return self.LastConv(last_conv_in)
        ########################################################

```

Question 2 Training curve for 25 epochs and a batch size of 100.



Question 3

(i) The coloured images are now more reasonable, the model with skip connections can now colour part of sky and grassland correctly.

(ii) Skip connections successfully help improve performances. The validation loss reduces from 1.83 to 1.35 (-26%), and the validation accuracy increases from 33% to 49%.

(iii) The quality of coloured images is improved significantly compared with outcome of the previous CNN without skip connections.

(iv) The model with skip connections has more trainable parameters than the previous CNN, it is more likely for it to pick up more complicated patterns.

(v) In the CNN without skip connections, the input to the very last convolution layer is an abstract representation of the original image. It could be that the representation is too abstract and the last convolution layer fails to infer colouring information from such an abstract representation. Adding skip connections helps the last convolution layer recall the previous information such as the raw image and activations of previous layers. In this case, the CNN with skip connections outperforms the vanilla CNN.

Question 4 The training/validation loss and final images are included below. And the table below summarizes the validation loss and accuracy after 25 epochs. In most coloured pictures from models trained with small batch sizes (50 and 100), the colouring is finer and more accurate especially at edges of objects. Evaluating metrics suggest a smaller batch size helps improve both models' losses and accuracy on the validation set.

Table 3: Model Performances with Different Batch Sizes

Batch Size	Validation Loss	Validation Accuracy
50	1.32	50%
100	1.36	49%
500	1.52	43%
1,000	1.63	41%

Figure 1: Final Image Outputs with Batch Size = 50, 100, 500, 1000 (from top to bottom)

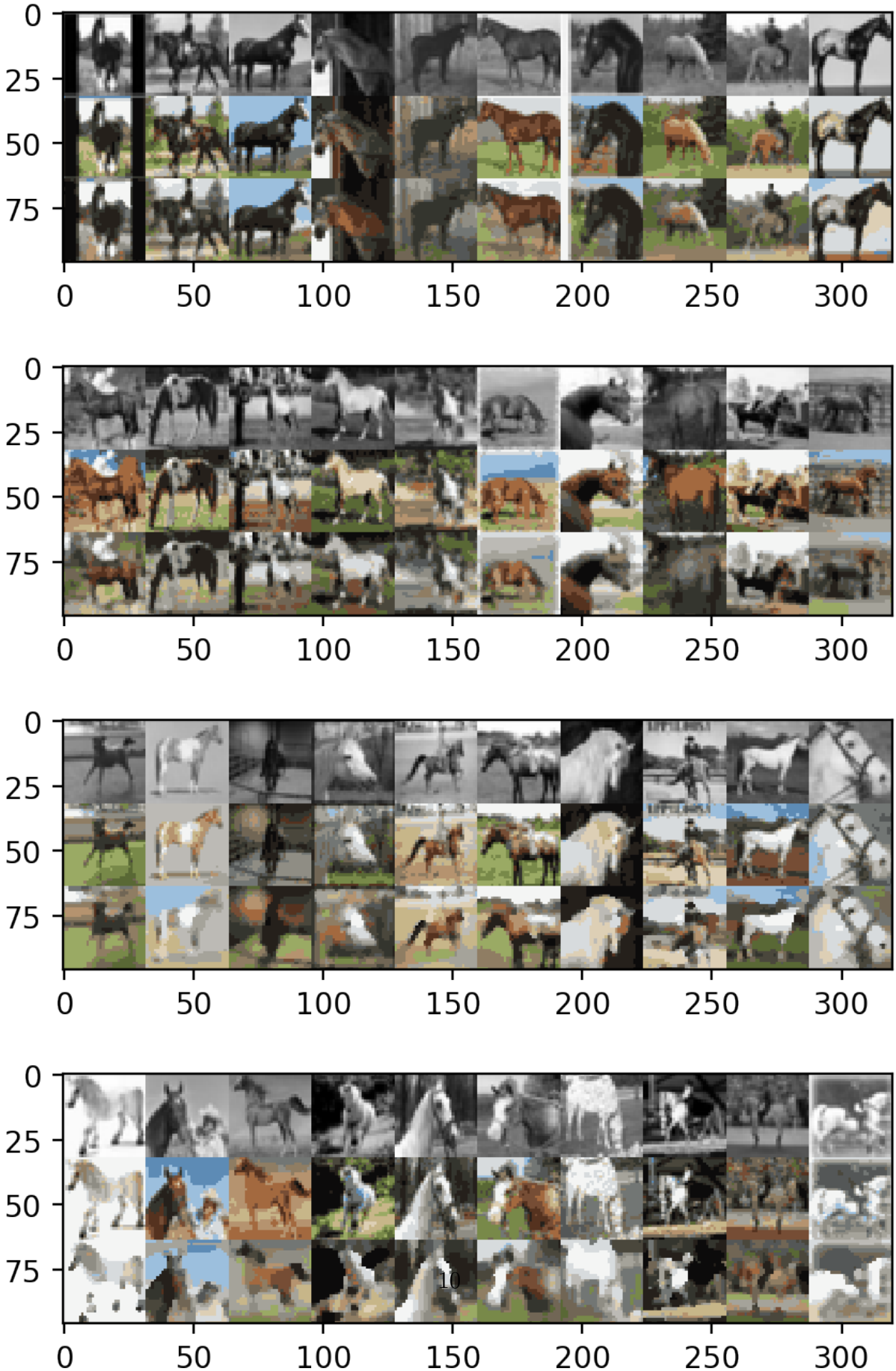


Figure 2: Batchsize=50(left) and 100(right)

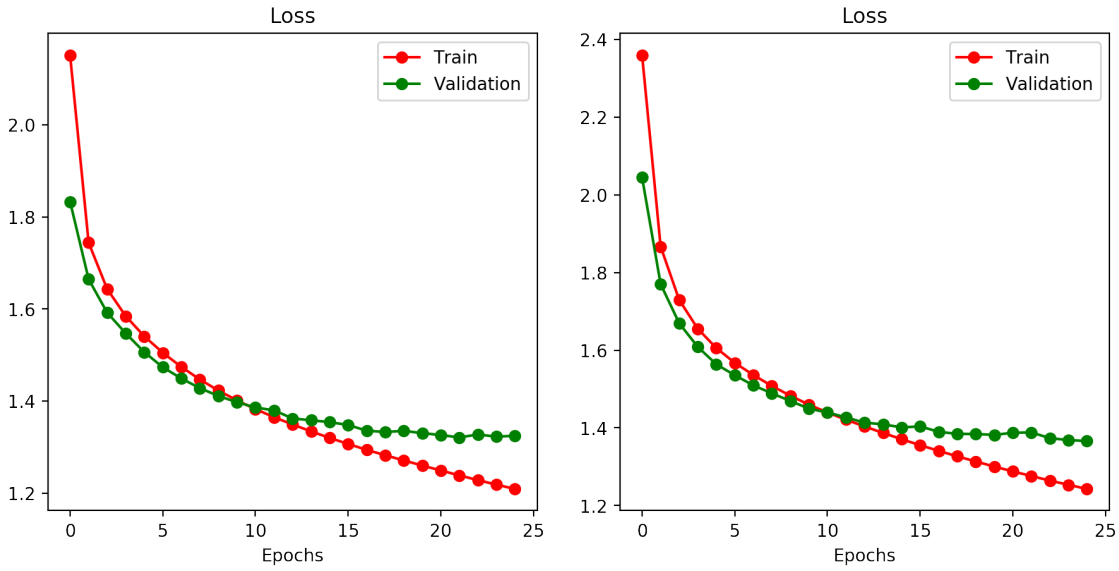
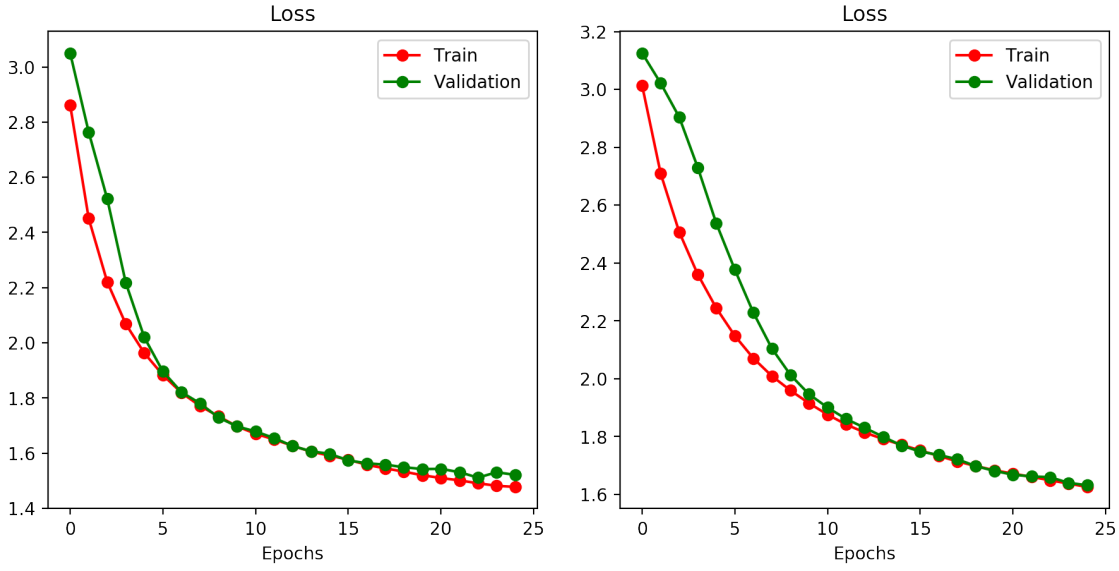


Figure 3: Batchsize=500(left) and 1,000(right)



3 Fine-tune Semantic Segmentation Model

Question 1 Implementation:

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

```

Question 2 Implementation:

```
model.requires_grad_(False)
model.classifier[4] = nn.Conv2d(
    256, 21, kernel_size=(1, 1), stride=(1, 1)
)

```

Question 3 The visualized predictions:

Figure 4: Prediction on Training Set

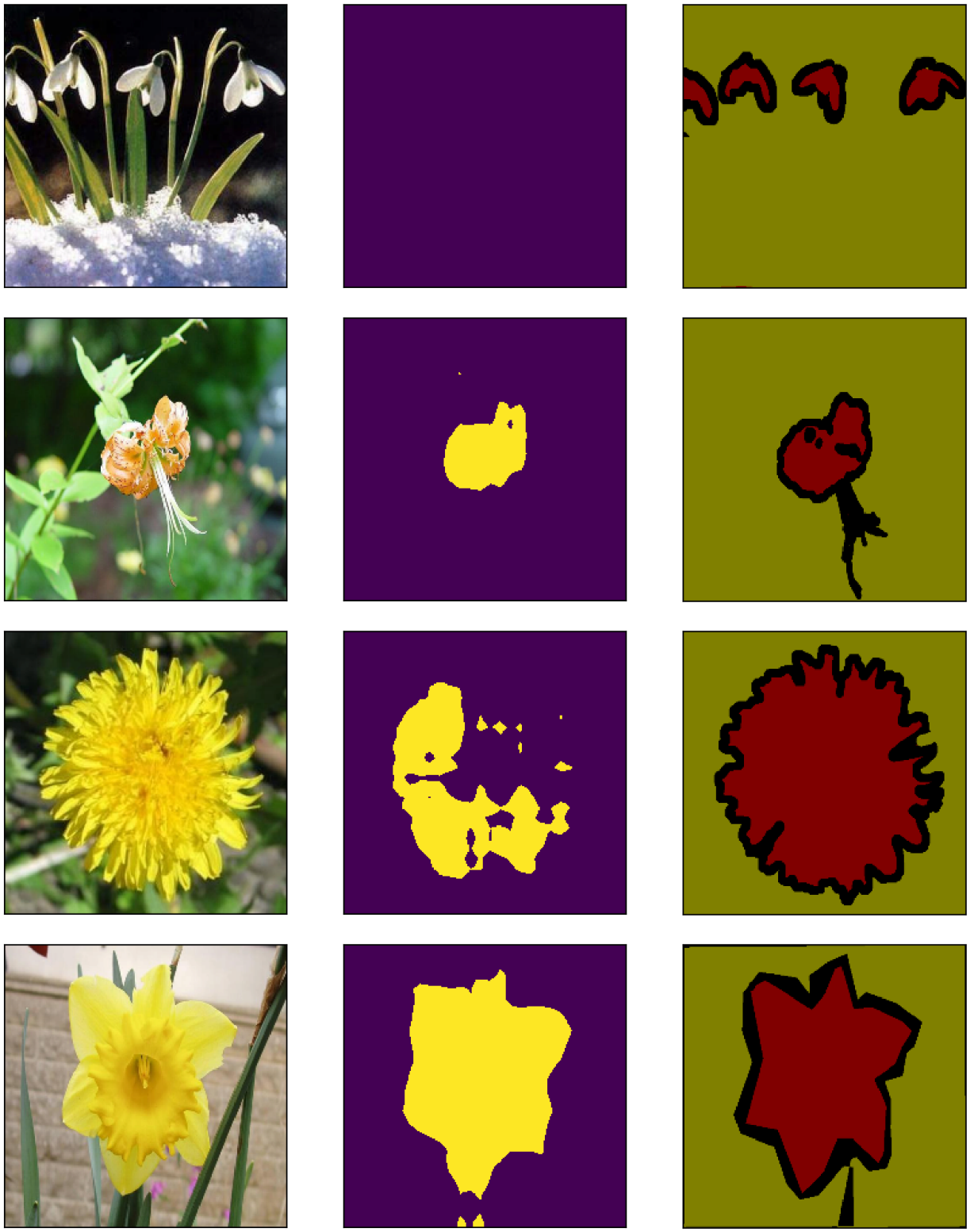


Figure 5: Prediction on Validation Set

Question 4 Memory Cost The memory required of tuning the entire model involves storing all trainable parameters and the gradient with respect to each trainable parameter. If we are tuning the entire model, the activation of all layers must be stored so that we can compute the error signal (gradient of loss w.r.t. weights in hidden layer) of each layer during backward phase. Activations of hidden layers and gradient (in backward phase) takes memory cost at $\mathcal{O}(n)$.

However, while tuning only the last layer, since we only need the error signal, only the activation of the last hidden layer is needed in the backward phase. Therefore, during the forward passing, the activation of the previous hidden layer can be discarded after the activation of current hidden layer is computed. The total memory cost of the forward phase is $\mathcal{O}(1)$. Similarly, we only need to store the gradient of loss w.r.t. weights in the last layer, the memory cost of backward phase is $\mathcal{O}(1)$ as well.

The overall memory complexity of tuning the entire model is in $\mathcal{O}(n)$, and tuning only one layer is in $\mathcal{O}(1)$.

Computational Cost While tuning only one layer, we only need computational power at $\mathcal{O}(1)$ to do backward propagation. Meanwhile, the computational cost of backward propagation for tuning the entire model is at $\mathcal{O}(n)$. However, in both cases, we need to conduct the forward passing at a computational cost at $\mathcal{O}(n)$. Hence, tuning the entire model and tuning only one layer both have computational complexity of $\mathcal{O}(n)$.

Question 5 The number of trainable parameters in each layer within this model only depends on kernel sizes, number of input channels and number of output channels. These numbers are unchanged when the size of input image changes. Therefore, the total number of parameters and memory needed to store the model is unchanged.

However, the new dataset with larger images now requires 400% memory, compared with the original dataset, to be stored.