Hacettepe University - Computer Engineering

Deep Learning CMP784 Homework 2

Convolutional Neural Networks

Student Name Ayça KULA
Student ID: 202237285

Fall 2021

1 PART A: Colourization as Classification

1.1 Complete the model CNN

From the given figure in the assignment we can see that the layers should be as:
Layer 1 : MyConv2D-NF + MaxPool + BatchNorm-NF + ReLU
Layer 2 : MyConv2D-2NF + MaxPool + BatchNorm-2NF + ReLU
Layer 3 : MyConv2D-2NF + BatchNorm-2NF + ReLU
Layer 4: MyConv2D-NF + Upsample + BatchNorm-NF + ReLU
Layer 5: MyConv2D-NC + Upsample + BatchNorm-NC + ReLU
Last Layer: MyConv2D-NC

In code 1 you can see the obtained CNN architecture.

```
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 ###############
        # Call hyper-parameters in order to obtain CNN
        self.num_in_channels = num_in_channels  # number of channels in the image
        self.kernel = kernel
        self.padding = padding
        # Variables num_filters and num_colours will be defined wrt
        # the  number of input/output layers
        self.num_filters = num_filters
        self.num_colours = num_colours

        # Obtain 5 layers as in figure.
        # Layer 1 --> MyConv2D-NF + MaxPool + BatchNorm-NF + ReLU
        self.layer1 = 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()
        ])
        # Layer 2 -->  MyConv2D-2NF + MaxPool + BatchNorm-2NF + ReLU
        self.layer2 = 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()
    ])
    # Layer 3 --> MyConv2D-2NF + BatchNorm-2NF + ReLU
    self.layer3 = 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()
    ])
    # Layer 4 --> MyConv2D-NF + Upsample + BatchNorm-NF + ReLU
    self.layer4 = 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), # scaling factor of 2 is used
        nn.BatchNorm2d(num_features=self.num_filters),
        nn.ReLU()
    ])
    # Layer 5 --> MyConv2D-NC + Upsample + BatchNorm-NC + ReLU
    self.layer5 = 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 Layer --> MyConv2D-NC
    self.LastConv = MyConv2d(
        in_channels=self.num_colours,
        out_channels=self.num_colours,
        kernel_size=self.kernel,
        padding=self.padding
    )
    ####################################################
```

Algorithm 1: CNN architecture in colab

Forward method has been implemented as in 7.

```
def forward(self, x):
    ############### YOUR CODE GOES HERE ###############
    layer_list = [
        self.layer1, self.layer2, self.layer3, self.layer4, self.layer5]
    for layer in layer_list:
        for lay in layer:
            x = lay(x)
    x = self.LastConv(x)
    return x
    ####################################################
```

Algorithm 2: Implementation of forward method

1.2 Do these results look good to you? Why or why not?

Before plotting the results, I had to modify the def(train) function under the class of AttrDict(dict) as seen in 3 due to python3 error. The error was related with converting python2 to python 3.

```
# LOAD THE COLOURS CATEGORIES
# encoding='bytes' is added due to error
colours = np.load(args.colours, allow_pickle=True, encoding='bytes')[0]
```

Algorithm 3: Solution for error of python 3

One of the figure from the results can be seen in 1. The results are very blurry and the parts in the figure cannot be identified clearly. Therefore, these results does not look good to me.

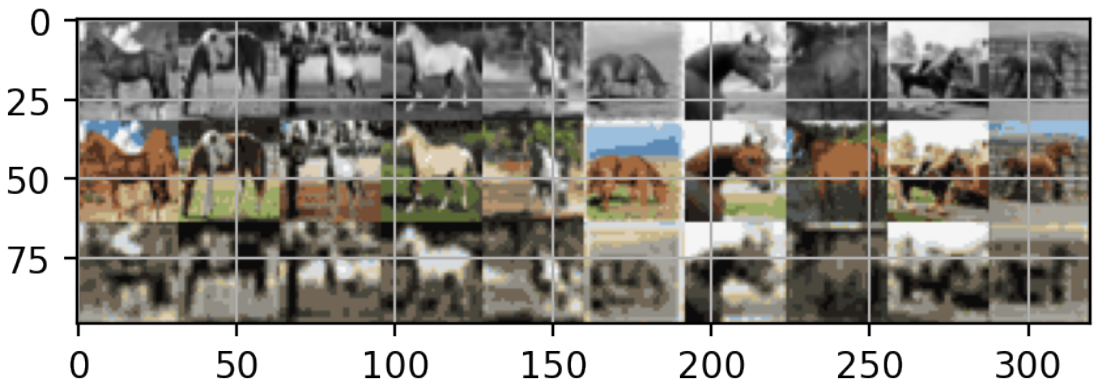

Figure 1: Iterative deepening search for the first 10x10 maze.

1.3 Compute the number of weights, outputs, and connections in the model

In CIFAR-10, images are size of 32x32x3 (32 width, 32 high, 3 color channels). And by using summary function imported from "torchsummary", the trainable parameters and output size could be seen and relation between all layers are investigated as seen in question 3 in colab file. The number of parameters are explained as:

- Convolutional Layer : Number of parameters in a CONV layer would be : ((shape of width of the filter * shape of height of the filter * number of filters in the previous layer+1)*number of filters). Where the term "filter" refer to the number of filters in the current layer.
- Batch Normalization layer: I believe that two parameters in the batch normalization layer are non-trainable(batch mean and batch standard deviation). Therefore number of params = (trainable params of batchNorm - untrainable params of batchNorm) x dimensionoflastaxis.
- Pool layer: This has got no learnable parameters because all it does is calculate a specific number, no backprop learning involved! Thus number of parameters = 0 [1].

Num. of Layers	Layers	Number of weights	Output Size	Connections
Layer 1	MyConv2D-NF	$k^2 \times NF + NF$	$32 \times 32 \times NF$	$32 \times 32 \times NF \times k^2$
	MaxPool	0	$16 \times 16 \times NF$	
	BatchNorm-NF	$2 \times NF$	$16 \times 16 \times NF$	
	ReLU	0	$16 \times 16 \times NF$	
Layer 2	MyConv2D-2NF	$k^2 \times NF \times 2NF + 2NF$	$16 \times 16 \times 2NF$	$16 \times 16 \times NF \times 2NF \times k^2$
	MaxPool	0	$8 \times 8 \times 2NF$	
	BatchNorm-2NF	$4 \times NF$	$8 \times 8 \times 2NF$	
	ReLU	0	$8 \times 8 \times 2NF$	
Layer 3	MyConv2D-2NF	$k^2 \times 2NF \times 2NF + 2NF$	$8 \times 8 \times 2NF$	$8 \times 8 \times 2NF \times 2NF \times k^2$
	BatchNorm-2NF	$4 \times NF$	$8 \times 8 \times 2NF$	
	ReLU	0	$8 \times 8 \times 2NF$	
Layer 4	MyConv2D-NF	$k^2 \times 2NF \times NF + NF$	$8 \times 8 \times NF$	$8 \times 8 \times 2NF \times NF \times k^2$
	Upsample	0	$16 \times 16 \times NF$	
	BatchNorm-NF	$2 \times NF$	$16 \times 16 \times NF$	
	ReLU	0	$16 \times 16 \times NF$	
Layer 5	MyConv2D-NC	$k^2 \times NF \times NC + NC$	$16 \times 16 \times NC$	$16 \times 16 \times NF \times NC \times k^2$
	Upsample	0	$32 \times 32 \times NC$	
	BatchNorm-NC	$2 \times NC$	$32 \times 32 \times NC$	
	ReLU	0	$32 \times 32 \times NC$	
Last Layer	MyConv2D-NC	$k^2 \times NC \times NC + NC$	$32 \times 32 \times NC$	$32 \times 32 \times NC \times NC \times k^2$

Table 1: 32x32 CNN weights, output, connections

If we sum up all the number of weights in 1 we get the result of total number of trainable parameters for CNN with size 32x32 as:

$$k^2\left(\mathrm{NC}^2+\mathrm{NCNF}+8\mathrm{NF}^2+\mathrm{NF}\right)+4\mathrm{NC}+18\mathrm{NF} \tag{1}$$

Moreover, the connections are:

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

Num. of Layers	Layers	Number of weights	Output Size	Connections
Layer 1	MyConv2D-NF	$k^2 \times NF + NF$	$64 \times 64 \times NF$	$64 \times 64 \times NF \times k^2$
	MaxPool	0	$32 \times 32 \times NF$	
	BatchNorm-NF	$2 \times NF$	$32 \times 32 \times NF$	
	ReLU	0	$32 \times 32 \times NF$	
Layer 2	MyConv2D-2NF	$k^2 \times NF \times 2NF + 2NF$	$32 \times 32 \times 2NF$	$32 \times 32 \times NF \times 2NF \times k^2$
	MaxPool	0	$16 \times 16 \times 2NF$	
	BatchNorm-2NF	$4 \times NF$	$16 \times 16 \times 2NF$	
	ReLU	0	$16 \times 16 \times 2NF$	
Layer 3	MyConv2D-2NF	$k^2 \times 2NF \times 2NF + 2NF$	$16 \times 16 \times 2NF$	$16 \times 16 \times 2NF \times 2NF \times k^2$
	BatchNorm-2NF	$4 \times NF$	$16 \times 16 \times 2NF$	
	ReLU	0	$16 \times 16 \times 2NF$	
Layer 4	MyConv2D-NF	$k^2 \times 2NF \times NF + NF$	$16 \times 16 \times NF$	$16 \times 16 \times 2NF \times NF \times k^2$
	Upsample	0	$32 \times 32 \times NF$	
	BatchNorm-NF	$2 \times NF$	$32 \times 32 \times NF$	
	ReLU	0	$32 \times 32 \times NF$	
Layer 5	MyConv2D-NC	$k^2 \times NF \times NC + NC$	$32 \times 32 \times NC$	$32 \times 32 \times NF \times NC \times k^2$
	Upsample	0	$64 \times 64 \times NC$	
	BatchNorm-NC	$2 \times NC$	$64 \times 64 \times NC$	
	ReLU	0	$64 \times 64 \times NC$	
Last Layer	MyConv2D-NC	$k^2 \times NC \times NC + NC$	$64 \times 64 \times NC$	$64 \times 64 \times NC \times NC \times k^2$

Table 2: 64x64 CNN weights, output, connections

If we sum up all the number of weights in 1 we get the result of total number of trainable parameters for CNN with size 64x64 as:

$$= k^2\left(8NF^2+NC^2+NFNC+NF\right)+18NF+4NC \tag{3}$$

Moreover, the connections are:

$$\begin{aligned} = k^2\big(&64^2NC^2+32^2NFNC+16^2\times 2NF^2+16^2\times 4NF^2+32^2\times 2NFNF \\ &+64^2NF\big) = k^2\left(4096NC^2+1024NFNC+5120NF^2+4096NF\right) \end{aligned} \tag{4}$$

1.4 Pre-processing step affect on output

Each pixel is multiplied by "a" and shifted by "b". Since the convolution operation is linear each weight is rearranged as:

$$\frac{\mathrm{w}-b}{a} \tag{5}$$

In this situation, previous questions have same colours compared with first results.

2 Skip Connections

2.1 Adding 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 ###############
        # Call hyper-parameters in order to obtain CNN
        self.num_in_channels = num_in_channels # number of channels in the image
        self.kernel = kernel
        # no padding

        # Variables num_filters and num_colours will be defined wrt
        # the  number of input/output layers
        self.num_filters = num_filters
        self.num_colours = num_colours

        # Obtain 5 layers as in figure.
        # Layer 1 --> MyConv2D-NF + MaxPool + BatchNorm-NF + ReLU
        self.layer1 = 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()
        )
        # Layer 2 -->  MyConv2D-2NF + MaxPool + BatchNorm-2NF + ReLU
        self.layer2 = 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()
```

```
)
# Layer 3 --> MyConv2D-2NF + BatchNorm-2NF + ReLU
self.layer3 = 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()
)

# Layer 4 --> MyConv2D-NF + Upsample + BatchNorm-NF + ReLU
self.layer4 = 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()
)
# Layer 5 --> MyConv2D-NC + Upsample + BatchNorm-NC + ReLU
self.layer5 = 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 Layer --> MyConv2D-NC
self.LastConv = MyConv2d(
    in_channels=self.num_colours + self.num_in_channels,
    out_channels=self.num_colours,
    kernel_size=self.kernel
)
#####################################################
```

Algorithm 4: "Adding skip connection" init method

```
def forward(self, x):
    ############### YOUR CODE GOES HERE ###############
    #  skip connection from the first layer to the last
    #  skip connection second layer to the second last
    layer1_out = self.layer1(x)
    layer2_out = self.layer2(layer1_out)
    layer3_out = self.layer3(layer2_out)
    # torch.cat --> concatenates the given sequence of seq tensors in the given
    layer4_in = torch.cat((layer2_out, layer3_out), dim=1)
    layer4_out = self.layer4(layer4_in)
    layer5_in = torch.cat((layer1_out, layer4_out), dim=1)
    layer5_out = self.layer5(layer5_in)
    last_conv_in = torch.cat((x, layer5_out), dim=1)

    return self.LastConv(last_conv_in)
    ###################################################
```

Algorithm 5: "Adding skip connection" forward method

2.2 Training curve

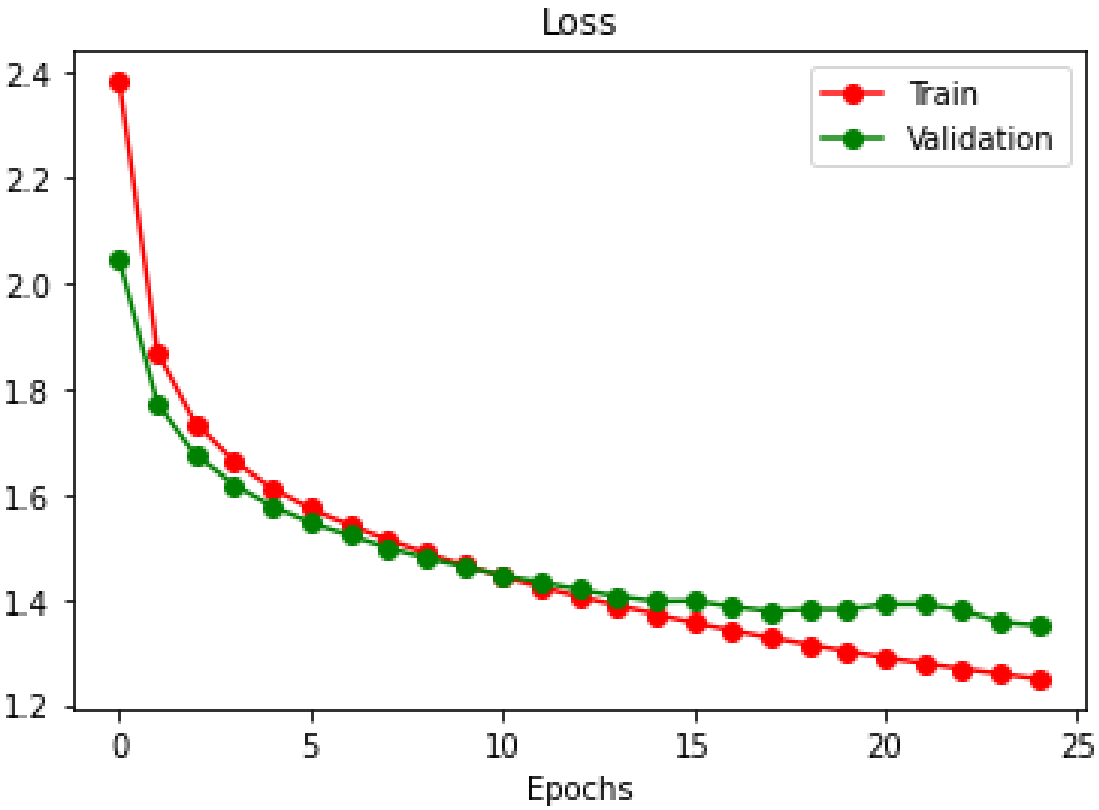

Figure 2: Training curve for 25 epochs and a batch size of 100.

2.3 Result compare to the previous model

1. How does the result compare to the previous model?
   The image is more clear compared to previous one. Especially, the sky and ground part of the figure is clearer as seen from 3.

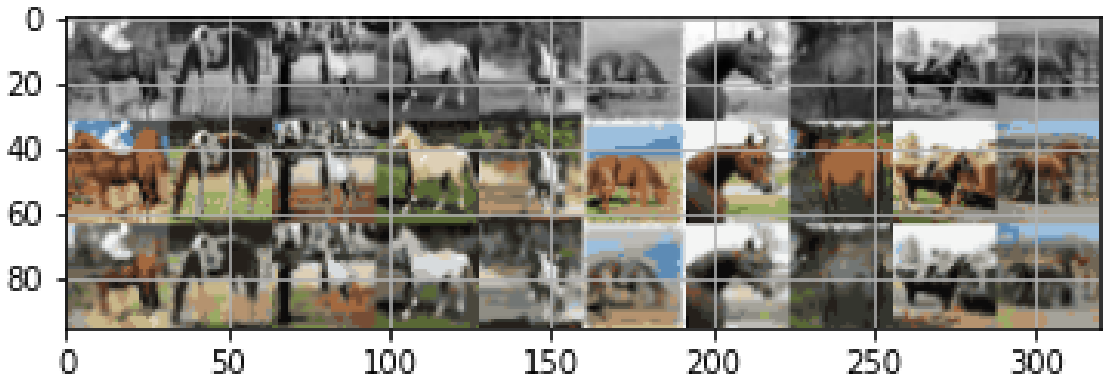

Figure 3: Training curve for 25 epochs and a batch size of 100.

2. Did skip connections improve the validation loss and accuracy?
   In the first part the validation value was oscillating however, in second part these oscillations were gone as seen in 2. For the first part, the loss started from 3.2 and dropped down to 2.1. However, in the second part the loss started from 2.4 and dropped down to 1.3. Therefore, we can say that the validation loss decreases by using skip connections. Also, the validation accuracy increases.

3. Did the skip connections improve the output qualitatively? How?
   Skip connections improved the quality of the output images.The feature information from input data is transmitted to the output more robustly with skip connections. The reason for first part to have lower quality may be that the greater the errors in the backproped prediction errors, the more meaningless the updates calculated for the learnable parameters in that layer.

4. Give at least two reasons why skip connections might improve the performance of our CNN models.
   Skip connections have an uninterrupted gradient flow from the first layer to the last layer, which tackles the vanishing gradient problem. By using a skip connection, we provide an alternative path for the gradient. It is experimentally validated that this additional paths are often beneficial for the model convergence [2]. Moreover, skip connections allow for feature reuse while also stabilizing training and convergence.

2.4 Analysis with different batch sizes

I tried to change batch sizes by keeping epoch constant to 25. Then I have recorded the validation loss and accuracy as given in table 3. I have seen from the graphs

obtained from colab that, as my batch size increases I increase my validation loss and decrease my validation accuracy.

Batch size	Validation Loss	Validation Accuracy
25	1.3284	49.8%
50	1.3236	49.6%
100	1.3469	49.2%
200	1.4001	47.4%
500	1.5225	43.1%
1000	1.6344	40.7%

Table 3: Validation loss and accuracy with different batch sizes.

Moreover, the images are clearer when small batch size is used as seen from figure 4 and 7.

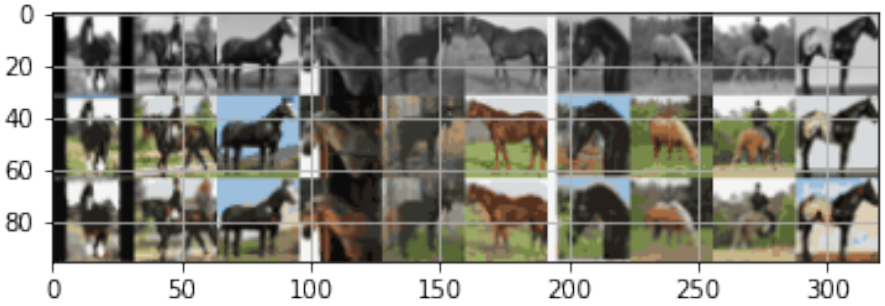

Figure 4: Image with batch size of 50.

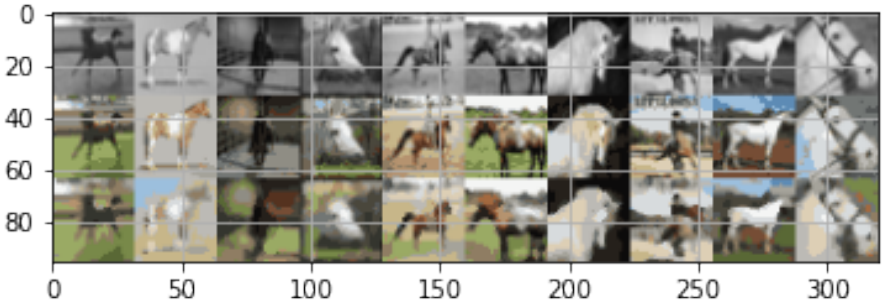

Figure 5: Image with batch size of 500.

3 Fine-tune Semantic Segmentation Model

3.1 Complete the train function

```
def train(args, model):

    # Set the maximum number of threads to prevent crash in Teaching Labs
    torch.set_num_threads(5)
    # Numpy random seed
    np.random.seed(args.seed)

    # Save directory
    # Create the outputs folder if not created already
    save_dir = "outputs/" + args.experiment_name
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

    learned_parameters = []
    # We only learn the last layer and freeze all the other weights
    ################ Code goes here ######################
    for name, weight in model.named_parameters():
        # The last layer weights have names prefix classifier.4
        if name.startswith("classifier.4"):
            # select the corr. weights then passing them to learned_parameters
            learned_parameters.append(weight)
    ######################################################
```

Algorithm 6: Completing train function

3.2 Complete the script

```
################ Code goes here ######################
# Around 2 lines of code is necessary
# to prevent back-prop through all the layers that should be frozen
model.requires_grad_(False)
# use nn.Conv2d
model.classifier[4] = nn.Conv2d(256, 21, kernel_size=(1, 1), stride=(1, 1))
######################################################
```

Algorithm 7: Completing script

3.3 Visualize the predictions

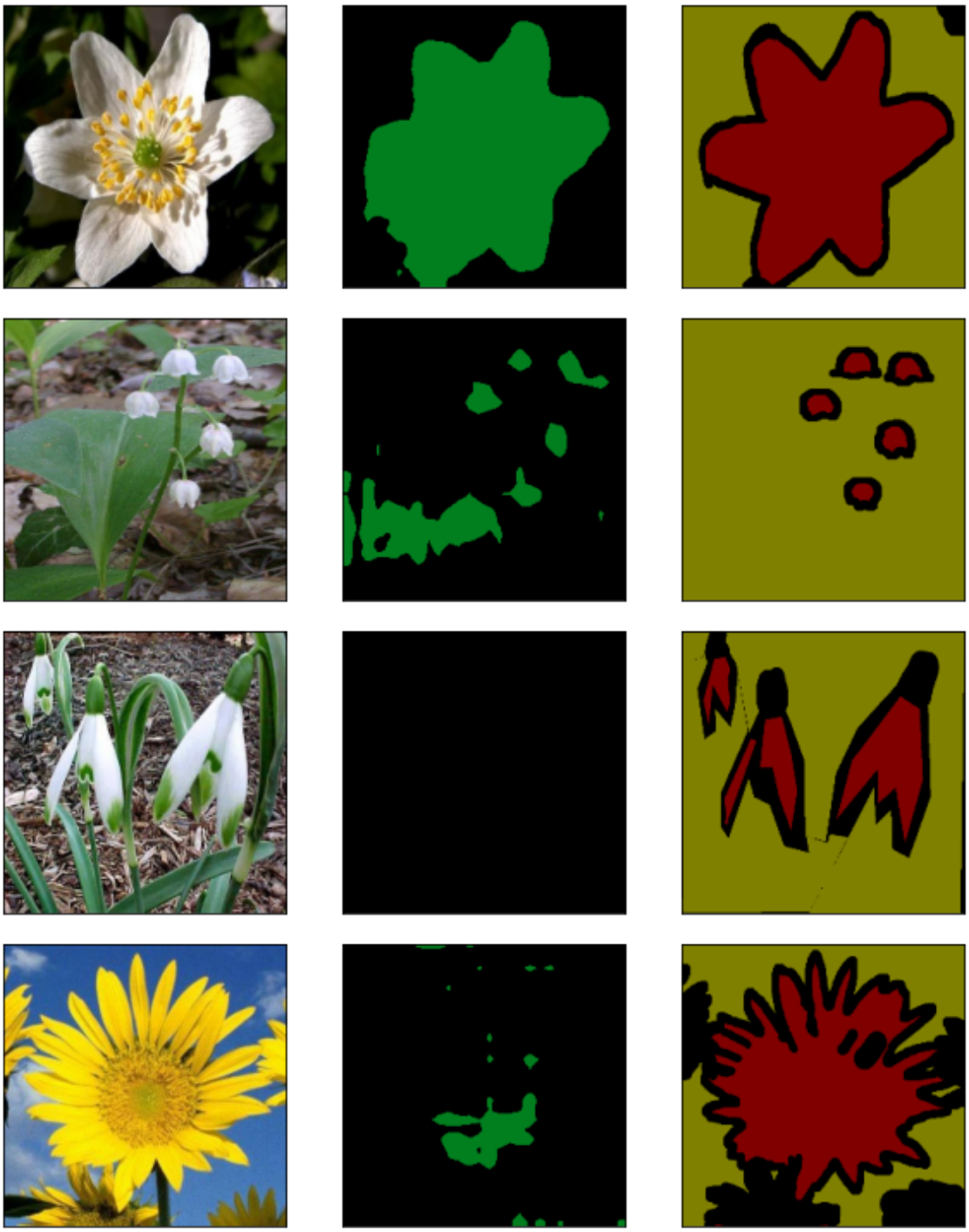

Figure 6: Visualize predictions.

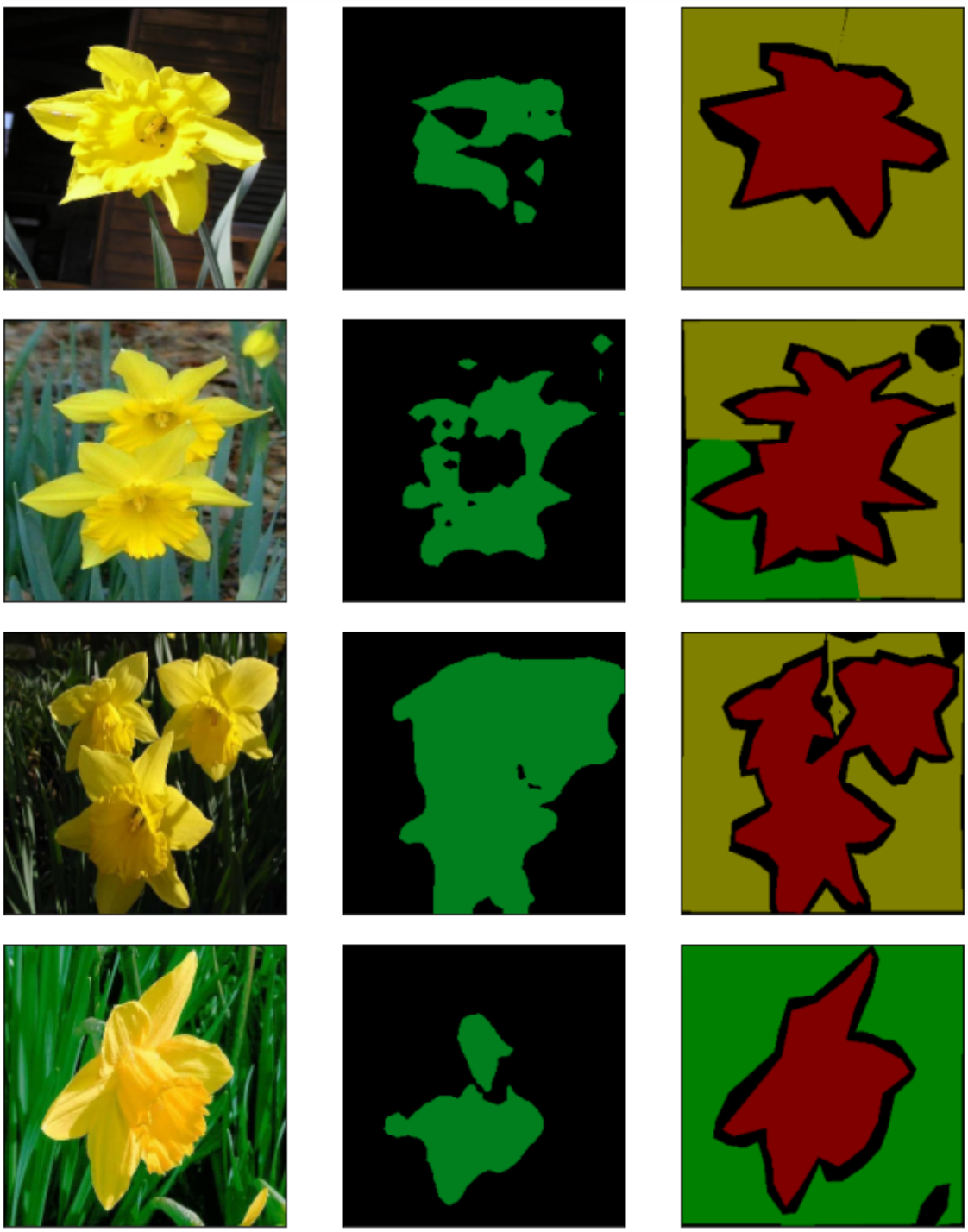

Figure 7: Visualize predictions.

3.4 Describe the difference in memory complexity

In order understand this question,we need to know what is being stored in GPU memory during training [3]:

- Parameters — The weights and biases of the network.
- Optimizer's variables — Per-algorithm intermediate variables (e.g. momentums).
- Intermediate calculations — Values from the forward pass that are temporarily stored in GPU memory and then used in the backward pass. (e.g. the activation outputs of every layer are used in the backward pass to calculate the gradients)
- Workspace — Temporary memory for local variables of kernel implementations.

NOTE: While first and fourth are always required, second and third are required only in training mode.

The memory complexity of tuning only one layer is O(1) and tuning the entire model is O(n). Since we have to store all trainable parameters for entire model the memory complexity is O(n). However, in one layer problem when computing the forward propagation part, the activation of the previous hidden layer is not computed since the current layer computation has been done.Loss, weights of last layer has been calculated [4]. However, we can say that tuning the entire model and tuning only one layer have the same computational complexity of O(n). Since, for one layer configuration we need to compute forward pass.

3.5 Increase the height and the width

This situation does not affect the memory complexity and the number of parameters of fine-tuning since the number of trainable parameters are unchanged.Since changing width or height of the input image has no affect on trainable parameter change.

References

[1] Rakshith Vasudev. Understanding and calculating the number of parameters in convolution neural networks (cnns), May 2020. URL https://towardsdatascience.com/understanding-and-calculating-the-number-of-parameters-in-convolution-neural-ne

[2] Nikolas Adaloglou. Intuitive explanation of skip connections in deep learning, Mar 2020. URL https://theaisummer.com/skip-connections/.

[3] Raz Rotenberg. How to break gpu memory boundaries even with large batch sizes, Jan 2020. URL https://towardsdatascience.com/how-to-break-gpu-memory-boundaries-even-with-large-batch-sizes-7a9c27a400ce.

[4] Kasper Fredenslund. Computational complexity of neural networks. URL https://kasperfred.com/series/introduction-to-neural-networks/computational-complexity-of-neural-networks.