# CSC413Coding_A2

haoyanjiang 1003324716

January 12, 2021

## 1 Part A: Colourization as Classification

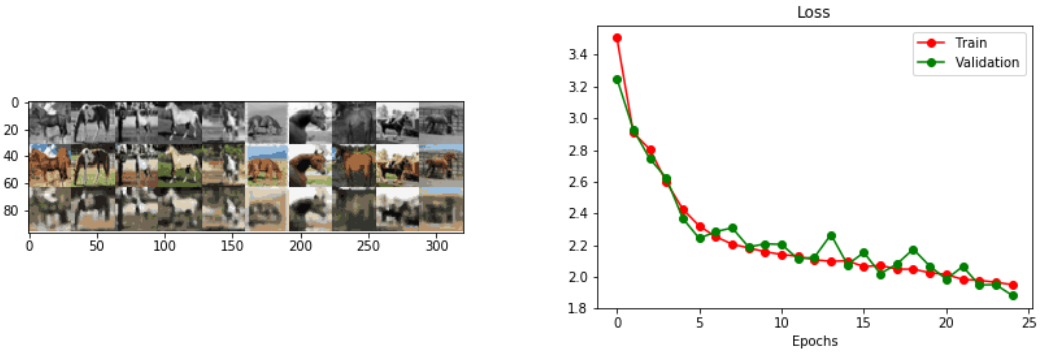
### 1.1

```
class CNN(nn.Module):
def __init__(self, kernel, num_filters, num_colours, num_in_channels):
    super(CNN, self).__init__()
    padding = kernel // 2
    self.model1 = nn.Sequential(
        MyConv2d(num_in_channels, num_filters, kernel_size=kernel, padding=padding),
        nn.MaxPool2d(2),
        nn.BatchNorm2d(num_filters),
        nn.ReLU()
    )
    self.model2 = nn.Sequential(
        MyConv2d(num_filters, num_filters*2, kernel_size=kernel, padding=padding),
        nn.MaxPool2d(2),
        nn.BatchNorm2d(2*num_filters),
        nn.ReLU()
    )
    self.model3 = nn.Sequential(
        MyConv2d(num_filters*2, num_filters*2, kernel_size=kernel, padding=padding),
        nn.BatchNorm2d(num_filters*2),
        nn.ReLU()
    )
    self.model4 = nn.Sequential(
        MyConv2d(num_filters*2, num_filters, kernel_size=kernel, padding=padding),
        nn.Upsample(scale_factor=2),
        nn.BatchNorm2d(num_filters),
        nn.ReLU()
    )
    self.model5 = nn.Sequential(
        MyConv2d(num_filters, num_colours, kernel_size=kernel, padding=padding),
        nn.Upsample(scale_factor=2),
        nn.BatchNorm2d(num_colours),
```

```
            nn.ReLU()
        )
        self.model6 = nn.Sequential(
            MyConv2d(num_colours, num_colours, kernel_size=kernel, padding=padding)
        )

    def forward(self, x):
        self.layer1 = self.model1(x)
        self.layer2 = self.model2(self.layer1)
        self.layer3 = self.model3(self.layer2)
        self.layer4 = self.model4(self.layer3)
        self.layer5 = self.model5(self.layer4)
        self.layer6 = self.model6(self.layer5)
        return self.layer6
```

## 1.2

**Running Results:**




(a) Image Colourization

(b) Loss for CNN against Epochs

Figure 1: Colorization with CNN

From Figure 1(a), we can see that the colorization for black white images is not authentic and realistic as targets.

In Figure 1(b), validation loss is not very stable, while the training error and validation error stay tuned indicates that the generalization for this model is pretty good

### 1.3

We have our convolution network with kernel size k = 3, image size W X H = 32 X 32, number of input channel = 1; for Batch Normalization, it normalize the image units, therefore it has not output units and connection, and each layer will have two weights: $\gamma$ and $\beta$, with 2×#channels-in weights in total

| layer | input size | # weights | # output units | # connections |
|---|---|---|---|---|
| Conv2D-NF | 32X32 | $3^2$×1×NF+NF | 32×32×NF | 32×32×$3^2$×1×NF |
| MaxPool2d(2) | 32X32 | 0 | 16×16×NF | 0 |
| BatchNorm2d-NF | 16X16 | 2NF | 0 | 0 |
| ReLU | 16X16 | 0 | 0 | 0 |
| Conv2D-2NF | 16X16 | $3^2$×NF×2NF+2NF | 16×16×2NF | 16×16×$3^2$×NF×2NF |
| MaxPool2d(2) | 8X8 | 0 | 8×8×2NF | 0 |
| BatchNorm2d-2NF | 8X8 | 4NF | 0 | 0 |
| ReLU | 8X8 | 0 | 0 | 0 |
| Conv2D-2NF | 8X8 | $3^2$×2NF×2NF+2NF | 8×8×2NF | 8×8×$3^2$×2NF×2NF |
| BatchNorm2d-2NF | 8X8 | 4NF | 0 | 0 |
| ReLU | 8X8 | 0 | 0 | 0 |
| Conv2D-NF | 8X8 | $3^2$×2NF×NF+NF | 8×8×NF | 8×8×$3^2$×2NF×NF |
| UpSample(2) | 8X8 | 0 | 16×16×NF | 0 |
| BatchNorm2d-NF | 16X16 | 2NF | 0 | 0 |
| ReLU | 16X16 | 0 | 0 | 0 |
| Conv2D-NC | 16X16 | $3^2$×NF×NC+NC | 16×16×NC | 16×16×$3^2$×NF×NC |
| UpSample(2) | 16X16 | 0 | 32×32×NC | 0 |
| BatchNorm2d-NC | 32X32 | 2NC | 0 | 0 |
| ReLU | 32X32 | 0 | 0 | 0 |
| Conv2D-NC | 32X32 | $3^2$×NC×NC+NC | 32×32×NC | 32×32×$3^2$×NC×NC |
| Total | | 27NF+4NC+72$NF^2$ +9NF×NC +9$NC^2$ | 2368NF+2304NC | 9216NF+8064$NF^2$ +2304NC×NF +9216$NC^2$ |

## 1.4

The linear transformation upon input images will not affect the final output. If this neural net is a CNN without batch normalization, the result will be scaled according to input scaling since CNN is linear. However, batch normalization layer performs internal co-variate shifts, which fixes the means and variances of each layer's inputs. This readjusts the input to new distributions and standardized to the same output. Therefore, the linear transformation will not affect the output.

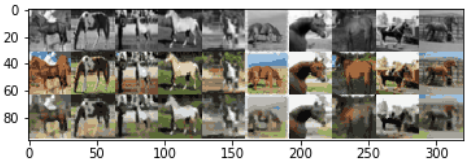
# 2 Part B: Skip Connections

## 2.1

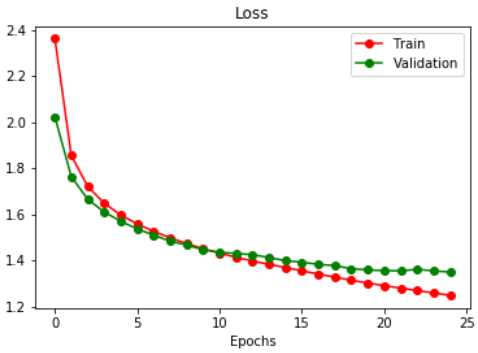
```
class UNet(nn.Module):
def __init__(self, kernel, num_filters, num_colours, num_in_channels):
    super(UNet, self).__init__()
    padding = kernel // 2
    self.model1 = nn.Sequential(
        MyConv2d(num_in_channels, num_filters, kernel, padding= padding),
        nn.MaxPool2d(2),
        nn.BatchNorm2d(num_filters),
        nn.ReLU()
    )
    self.model2 = nn.Sequential(
        MyConv2d(num_filters, num_filters*2, kernel, padding= padding),
        nn.MaxPool2d(2),
        nn.BatchNorm2d(2*num_filters),
        nn.ReLU()
    )
    self.model3 = nn.Sequential(
        MyConv2d(num_filters*2, num_filters*2, kernel, padding= padding),
        nn.BatchNorm2d(num_filters*2),
        nn.ReLU()
    )
    self.model4 = nn.Sequential(
        MyConv2d(num_filters*4, num_filters, kernel, padding= padding),
        nn.Upsample(scale_factor=2),
        nn.BatchNorm2d(num_filters),
        nn.ReLU()
    )
    self.model5 = nn.Sequential(
        MyConv2d(num_filters*2, num_colours, kernel, padding= padding),
        nn.Upsample(scale_factor=2),
        nn.BatchNorm2d(num_colours),
        nn.ReLU()
    )
    self.model6 = nn.Sequential(
        MyConv2d(num_colours+num_in_channels, num_colours, kernel, padding= padding)
    )

def forward(self, x):
    self.layer1 = self.model1(x)
    self.layer2 = self.model2(self.layer1)
    self.layer3 = self.model3(self.layer2)
```

```
        self.layer4 = self.model4(torch.cat((self.layer3, self.layer2), dim=1))
        self.layer5 = self.model5(torch.cat((self.layer4, self.layer1), dim=1))
        self.layer6 = self.model6(torch.cat((self.layer5, x), dim=1))
        return self.layer6
```

## 2.2




(a) Image Colourization




(b) Loss for CNN against Epochs

Figure 2: Colorization with CNN

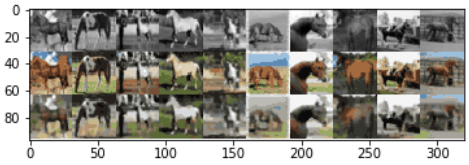
## 2.3

Compared with previous model, skip connections did a better job in general. As shown in Figure (a), we can see that the prediction is more similar to the expect value, and did a better coloring job. Although each training epoch takes slightly longer than previous model about half second, the loss function is much smaller, and with almost 1.5 larger the accuracy in general.Also can be seen in Figure (b) that the loss function curve is smooth, and validation error is close to training error indicates a good generalization of this model.

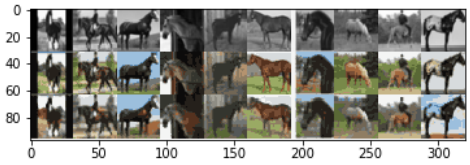
This may because that 1. some information that was captured in the first few layers was required for reconstruction. If we would not have used the skip architecture that information would have been lost. Therefore, we use skip architecture to feed information from previous layers explicitly to the later layers. 2. also, adding skip connections can skip layer that does not contribute to higher value or accuracy, therefore preserve the correct features and pull up accuracy at the same time.

**2.4**

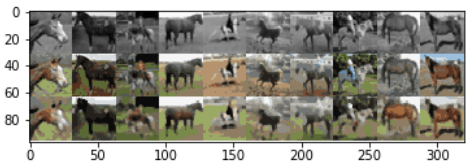
| batch size | training loss | validation loss | output quality ranking |
|---|---|---|---|
| 200 | 1.3424 | 1.4289 | 5 |
| 100 | 1.2483 | 1.3498 | 4 |
| 50 | 1.1980 | 1.3341 | 3 |
| 20 | 1.1740 | 1.3394 | 1 |
| 10 | 1.1674 | 1.3262 | 2 |



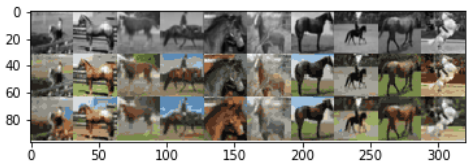

(a) Image Colourization with Batch Size of 100



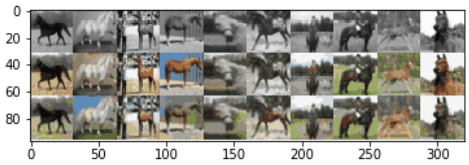

(b) Image Colourization with Batch Size of 50




(c) Image Colourization with Batch Size of 200




(d) Image Colourization with Batch Size of 20




(e) Image Colourization with Batch Size of 10

Figure 3: CNN output with different batch size

As we can see from table and Figure 3 that the smaller the batch size is, the smaller validation/-training error is. Also, the smaller batch size is, the better colorization this model does, except when batch size is 20, it is better than batch size 10.This may caused by the over-fitting of model or the data being trained is not representative such that it jeopardizes the generalization of this model.

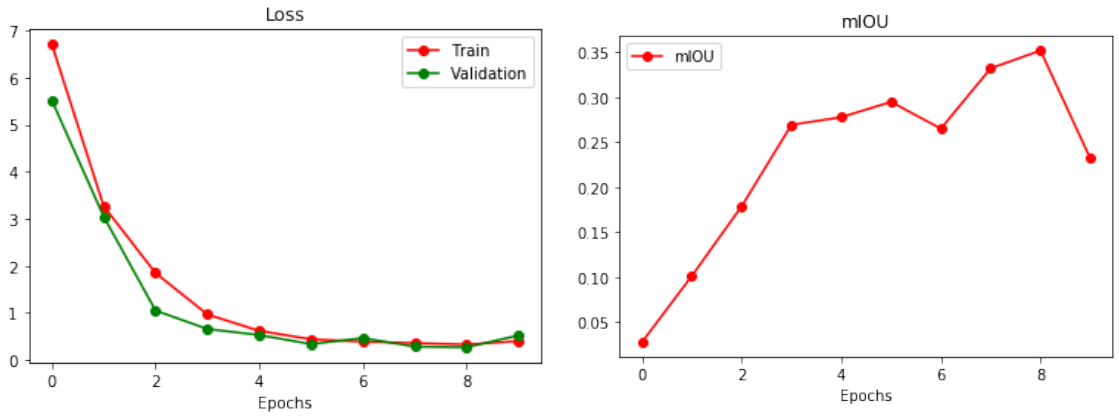
# 3 Part C: Fine-tune Semantic Segmentation Model

## 3.1

```
################ Code goes here ######################
for name, param in model.named_parameters():
  if name.startswith("classifier.4"):
    learned_parameters.append(param)
# Around 2-3 lines of code
#####################################################
```

## 3.2

```
################ Code goes here ######################
# Around 2 lines of code
model.requires_grad_(False)
model._modules['classifier'][4] = nn.Conv2d(256, 2, (3, 3))
#####################################################
```




(a) Loss for Training and Validation

(b) mIOU segmentation detection

Figure 4: Running results from fine-tuned model

### 3.3

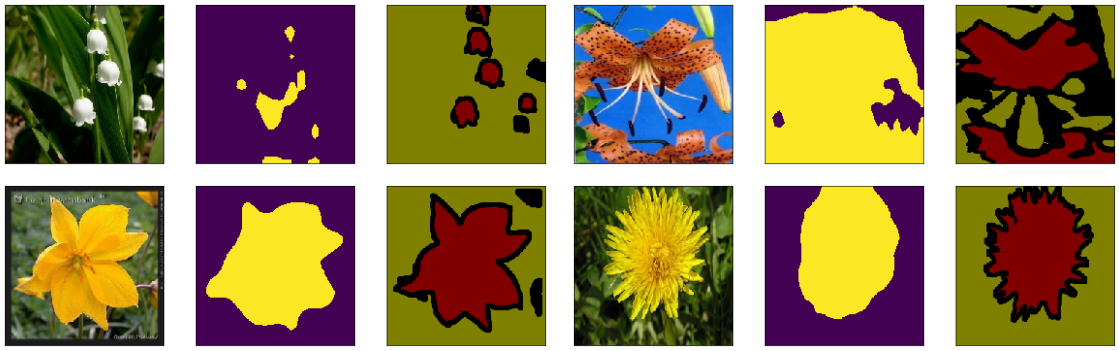


Figure 5: Running results from training input

Figure 6: Running results from validation input

### 3.4

For both models, memory is needed for forward training results relating to number of parameters. Assume each layer has similar amount of training results and parameters, then fine tuning only requires to train and remember one layer's data, while training the entire pre-trained model requires to memorize n layers' parameters. Therefore, training entire model requires memory storage O(n), while fine-tuning models requires O(1).
For computation complexity, since each layer has similar amount of parameters, and the computation logits are proportional to number of output units, the computation complexity of training entire model also requires computation power O(n), while fine-tuning models requires O(1).

### 3.5

The numbers of parameters will not be affects, since the weight and bias are only proportional to kernel size, input channels and output channels. So does the memory complexity, since memory is only taken by forward training to keep track of variables. If the number of variables does not affected by W and H, the complexity of memory will stay unchanged as well.