

EE488 Machine Learning Basics and Practices Project2

20170616 정희진

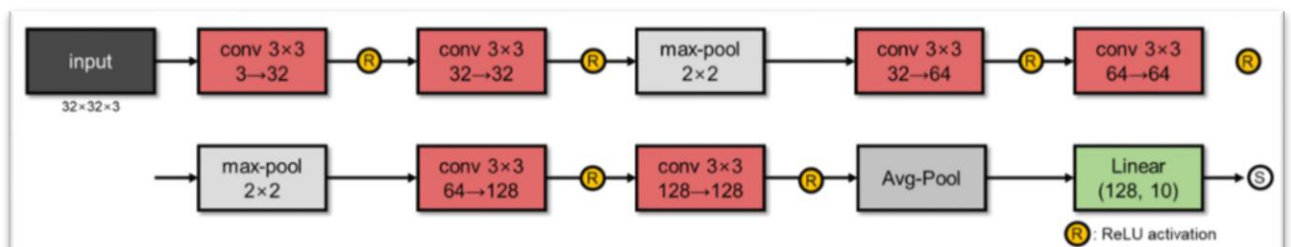
Part 1. Implement the Batchnormalization Layer in Convolutional Neural Network

Step 2: Run VGG without Batchnorm

Step 2-1. forward function

```
def forward(self, x):  
    ##### blank #####  
    x = self.conv1(x)  
    x = F.relu(x)  
    x = self.conv2(x)  
    x = F.relu(x)  
    x = self.max_pool(x)  
  
    x = self.conv3(x)  
    x = F.relu(x)  
    x = self.conv4(x)  
    x = F.relu(x)  
    x = self.max_pool(x)  
  
    x = self.conv5(x)  
    x = F.relu(x)  
    x = self.conv6(x)  
    x = F.relu(x)  
    #####  
  
    x = self.avg_pool(x)  
    x = x.view(-1, 128)  
    x = self.fc(x)  
  
    return x
```

왼쪽의 blank code는 그 아래의 그림을 그대로 코드로 나타낸 것이다. 코드에 나타난 x 는 아래 그림에서 네모 박스(conv, pool)에 들어가는 input을 나타낸다. 각각의 conv#, max_pool은 위의 코드에는 나와있지 않지만 이미 skeleton code(_init_ function)에 자세히 나타나 있다. input의 channel의 개수, output의 channel의 개수, kerner size, padding을 얼마만큼 할 것인지가 함수안에 나타나 있다.



Step 2-2. train function

```
def train(model, data_loader, criterion, optimizer, n_epoch):
    model.train()
    for epoch in range(n_epoch):
        running_loss = 0
        for i, (images, labels) in enumerate(data_loader):
            images, labels = images.cuda(), labels.cuda()
            ##### blank #####
            optimizer.zero_grad()

            outputs = model(images)
            loss = criterion(input=outputs, target=labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            #####

        print('Epoch {}, loss = {:.3f}'.format(epoch + 1, running_loss/len(data_loader)))
```

Blank code를 하나씩 살펴보자.
optimizer.zero_grad 함수를 통해 gradient를 초기화시켜준다. 그 다음으로 데이터의 output을 구하고 이를 실제값(label)과 비교해 loss를 구한다. loss.backward 함수를 통해 backpropagation을 진행하고 optimizer.step 함수를 통

해 weight와 같은 parameter들을 update한다. 각각의 loop마다 loss를 축적해 최종 loss를 구한다.

Step 2-3. eval function

```
def eval(model, data_loader):
    model.eval()
    total = 0
    correct = 0
    with torch.no_grad():
        for images, labels in data_loader:
            images, labels = images.cuda(), labels.cuda()
            ##### blank #####
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted==labels).sum().item()
            #####
    accuracy = 100*correct/total

    print('Test Accuracy: {}'.format(accuracy))
```

eval function은 train된 model에 input data를 넣고 output값과 실제값(label)을 비교하여 model이 얼마만큼의 정확도를 갖는지 퍼센트로 나타낸 것이다.

Step 2-4. train result

```
Epoch 1, loss = 1.768
Epoch 2, loss = 1.357
Epoch 3, loss = 1.153
Epoch 4, loss = 1.021
Epoch 5, loss = 0.923
Epoch 6, loss = 0.849
Epoch 7, loss = 0.785
Epoch 8, loss = 0.744
Epoch 9, loss = 0.694
Epoch 10, loss = 0.655
```

epoch값이 커질수록(train을 더 많이 할수록) loss값이 작아진다.

Step 2-5. test result

Test Accuracy: 75.35%

Step 3: Implement MyBatchNorm2d() class inheriting nn.Module

```
class MyBatchNorm2d(nn.Module):
    def __init__(self, output_num_channel):
        super(MyBatchNorm2d, self).__init__()
        self.gamma = nn.Parameter(torch.tensor(1.0)) # register the tensor as a parameter in this module (be treated like module's parameter -> can be learned via optimizer altogether)
        self.beta = nn.Parameter(torch.tensor(0.0))

        ##### blank #####
        self.moving_mean = 0.0
        self.moving_var = 0.0
        self.t = 1
        #####

    def forward(self, input):
        ##### blank #####
        if self.training: # set to be True automatically when 'model.train()' is called
            mean = torch.sum(input, dim=0) / input.size(dim=0)
            var = torch.sum((input - mean)**2, dim=0) / input.size(dim=0)
            norm_input = (input - mean) / torch.sqrt(var+1e-8)
            output = self.gamma * norm_input + self.beta

            if self.t == 1:
                self.moving_mean = mean
                self.moving_var = var
                self.t = self.t + 1
            else:
                self.moving_mean = 0.1*mean + 0.9*self.moving_mean
                self.moving_var = 0.1*var + 0.9*self.moving_var
                self.t = self.t + 1

        else: # if the 'model.eval()' was called
            norm_input = (input - self.moving_mean) / torch.sqrt(self.moving_var+1e-8)
            output = self.gamma * norm_input + self.beta

        #####

        return output
```

Input: Values of x over a mini-batch: $B = \{x_1, \dots, x_m\}$;
 Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

The EMA for a series Y may be calculated recursively:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha Y_t + (1 - \alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

Where:

- The coefficient α represents the degree of weighting decrease, a constant smoothing factor between 0 and 1. A higher α discounts older observations faster.
- Y_t is the value at a time period t .
- S_t is the value of the EMA at any time period t .

Blank code를 설명하기 위해 각각의 변수를 살펴보자. 먼저 forward 함수에서 model이 training 진행중에 있다면 변수는 다음과 같이 대응된다.

Variable in code	Symbol
mean	μ_B
var	σ_B^2
norm_input	\hat{x}
output	y
self.gamma	γ
self.beta	β

이때, `__init__` 함수 안에서 `self.gamma`와 `self.beta`의 초기값은 각각 1과 0으로 설정되었다. 또한 `mean`과 `var` 변수를 구할 때 함수 안 parameter로 `dim=0`이 사용되었는데, 이는 `input.shape[0]` 값이 batch의 크기(m)를 나타내기 때문이다. `norm_input`의 분모 부분은 0이되지 않도록 아주 작은 값($1e-8$)을 추가로 더해주었다.

다음으로 train된 model에 test 데이터를 넣어 evaluation을 진행한다고 하자. 그렇다면 train 과정에서 계산된 `mean`과 `var`을 이용해 batch normalization 해야 할 것이다. 대응되는 변수는 다음과 같다.

Variable in code	Symbol
mean (or var)	Y_t
<code>self.mov_mean</code> (or <code>self.mov_var</code>)	S_t
<code>self.t</code>	t

`__init__` 함수에서 먼저 각각의 변수를 선언해주었고 특히, `self.t`의 초기값은 1로 설정하였다. `forward` 함수의 training 부분에서 `self.t`가 1일 때와 아닐 때를 구분하여 exponential moving average 방법을 이용해 `self.mov_mean`값과 `self.mov_var`값을 구하였다. 그리고 그 값들을 이용해 evaluation 부분에서 training 부분과 비슷하게 (`mean`과 `variation` 변수만 다르게) batch normalization을 하였다.

Step 4: Implement VGG with MyBatchNorm2d()

Step 4-1. forward function

```
def forward(self, x):  
    ##### blank #####  
    x = self.conv1(x)  
    x = self.norm1(x)  
    x = F.relu(x)  
    x = self.conv2(x)  
    x = self.norm2(x)  
    x = F.relu(x)  
    x = self.max_pool(x)  
  
    x = self.conv3(x)  
    x = self.norm3(x)  
    x = F.relu(x)  
    x = self.conv4(x)  
    x = self.norm4(x)  
    x = F.relu(x)  
    x = self.max_pool(x)  
  
    x = self.conv5(x)  
    x = self.norm5(x)  
    x = F.relu(x)  
    x = self.conv6(x)  
    x = self.norm6(x)  
    x = F.relu(x)  
    #####  
  
    x = self.avg_pool(x)  
    x = x.view(-1, 128)  
    x = self.fc(x)  
    return x
```

이 부분은 Step 2-1과 비슷하다. 추가적으로, batch normalization을 하기 위해 `x = self.norm#(x)`과 같은 코드가 중간에 추가되었다.

Step 4-2. train result

```
Epoch 1, loss = 1.351  
Epoch 2, loss = 0.989  
Epoch 3, loss = 0.847  
Epoch 4, loss = 0.759  
Epoch 5, loss = 0.695  
Epoch 6, loss = 0.647  
Epoch 7, loss = 0.614  
Epoch 8, loss = 0.579  
Epoch 9, loss = 0.559  
Epoch 10, loss = 0.531
```

epoch값이 커질수록(train을 더 많이 할수록) loss값이 작아진다.

Step 4-3. test result

Test Result with Batch Normalization	Test Result without Batch Normalization
Test Accuracy: 80.56%	Test Accuracy: 75.35%

Batch normalization을 한 model의 test accuracy가 더 높다.

Part2. Neural Style Transfer

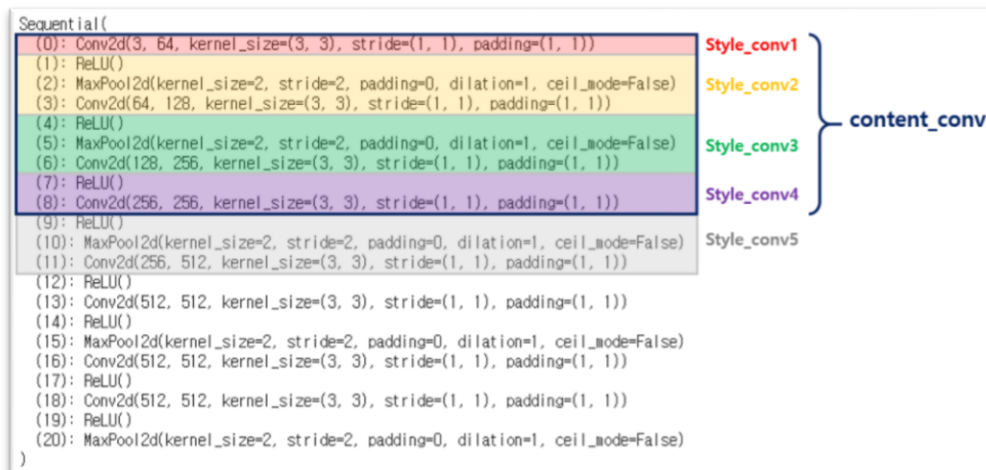
Step 2. Compute style loss and content loss

Step 2-3. content_conv and style_conv

```
##### blank #####
content_conv = cnn[0:9]
style_conv1 = cnn[0:1]
style_conv2 = cnn[1:4]
style_conv3 = cnn[4:7]
style_conv4 = cnn[7:9]
style_conv5 = cnn[9:12]
#####

style_convlist = [style_conv1, style_conv2, style_conv3, style_conv4, style_conv5]
```

아래의 그림은 cnn을 나타낸 것이다. 그림에 표시된 대로, cnn을 slicing 하여 각각의 변수를 만들었다.



Step 2-4.

< mse_loss function >

```
def mse_loss(input, target):
    ##### blank #####
    loss = torch.sum((input - target)**2)
    return loss
    #####
```

$$\text{loss} = \sum (\text{input} - \text{target})^2$$

왼쪽의 함수는 오른쪽 식을 나타낸다. 이 때, 다음에 나타날 함수 (compute_loss 함수)에서 더 편리하게 사용하기 위해 torch.mean 함수가 아니라 torch.sum 함수를 사용하였다.

< gram_matrix function >

```
def gram_matrix(feature_map):  
    ##### blank #####  
    depth = feature_map.shape[1]  
    width = feature_map.shape[2]  
    height = feature_map.shape[3]  
  
    vec_feature_map = torch.reshape(feature_map, [depth, -1])  
    gram = torch.matmul(vec_feature_map, vec_feature_map.T) / (width * height * depth)  
    return gram  
#####
```

Feature_map의 depth, width, height을 뽑아낸 뒤, 각각의 feature map을 기준으로 재배열시켰다(vec_feature_map). 그런 다음 vec_feature_map 행렬과 transpose된 행렬을 dot

product시키면 gram matrix가 생성된다. 이 때 이후의 계산(compute_loss 함수)을 편리하게 하기 위해 depth, width, height로 나누어 주었다. Gram matrix의 element가 어떻게 이루어져있는지는 lecture note에 자세히 나타나있다.

let a matrix G^l collect the dot products between the vectorized (2D→1D) feature maps i and j

< compute_content_loss >

```
def compute_content_loss(input, target, content_conv):  
    ## Compute the mse-based content loss using content_conv  
  
    ##### blank #####  
    x = content_conv(input)  
    y = content_conv(target)  
    loss = mse_loss(x, y) / 2  
    return loss  
#####
```

$$J^{content} = \frac{1}{2} \sum_{j,k} (V_{jk}^c - V_{jk}^g)^2$$

Content loss는 오른쪽 식과 같다. 여기서 V_{jk}^c 는 convolution layer의 input이 content image 일 때, output의 j번째 feature map의 k번째 pixel position을 나타낸다(V_{jk}^g 는 generated image). 위에서 구한 mse_loss함수를 이용해서 쉽게 구할 수 있다.

< compute_style_loss >

```
def compute_style_loss(input, target, style_convlist):
    ## Compute the ase-based style loss using gram_matrix and style_convlist

    ##### blank #####
    x = input
    y = target
    loss = 0

    for style_conv in style_convlist:
        x = style_conv(x)
        y = style_conv(y)
        gram_x = gram_matrix(x)
        gram_y = gram_matrix(y)
        loss = loss + mse_loss(gram_x, gram_y) / 4
    return loss
#####
```

$$J^l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - S_{ij}^l)^2$$
$$J^{style} = \sum_{l=0}^L J^l$$

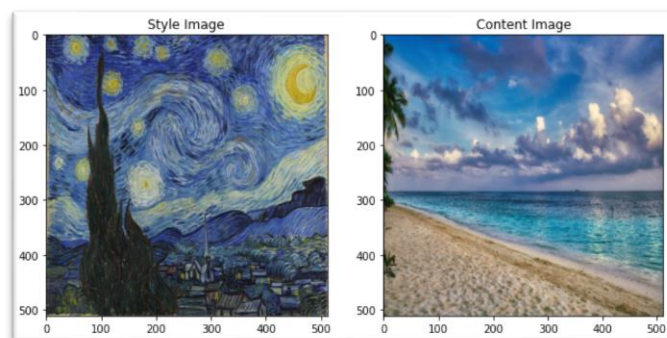
Style loss는 오른쪽 식과 같다. 여기서 G^l 은 convolution layer의 input이 generated image 일 때, output의 gram matrix를 나타낸다(S^l 는 style image). 앞에 곱해진 $N_l^2 M_l^2$ 은 output의 (depth * width * height)²을 나타내고 이는 이미 gram_matrix 함수 안에서 계산되었으니 4로 나누어 준다. 그리고 style_convlist에 있는 모든 convolution layer의 output을 이용해 구한 각각의 loss를 총합하면 style loss를 구할 수 있다.

Step 3: Training

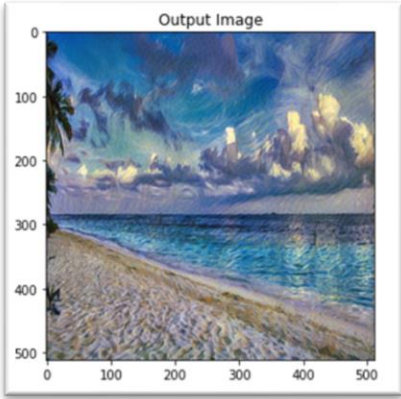
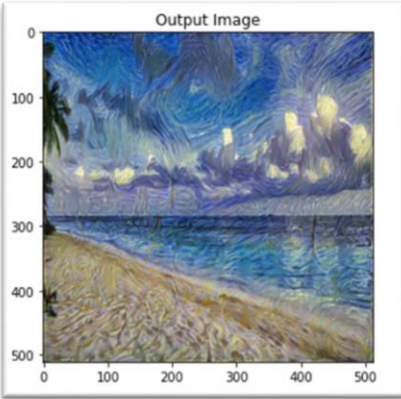
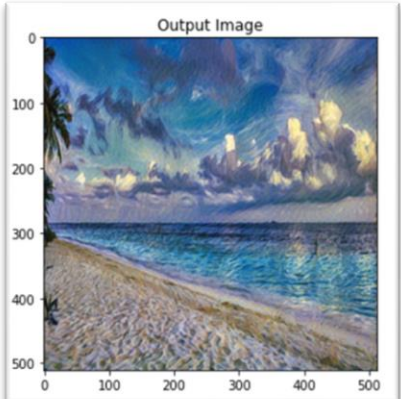
Step 3에서 총 loss를 구하는 코드는 다음과 같다.

```
content_loss = compute_content_loss(input_img, content_img, content_conv)    ## compute content loss
style_loss = compute_style_loss(input_img, style_img, style_convlist)        ## compute style loss
loss = content_weight * content_loss + style_weight * style_loss            ## compute total loss
```

Content image와 style image는 다음과 같다.



style weight와 content weight의 값에 따라 output image는 다음과 같이 나왔다.

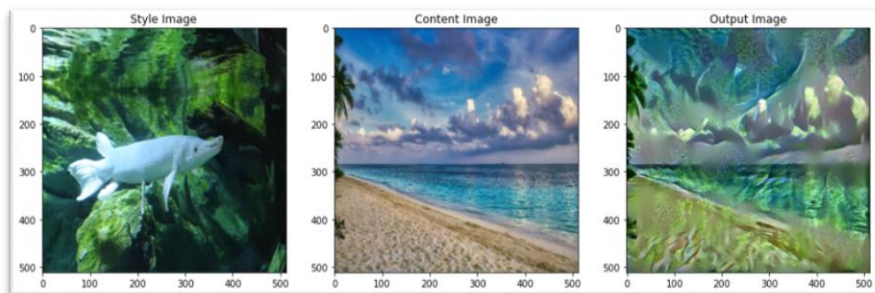
$w_s = 10^7, w_c = 1$	$w_s = 10^{12}, w_c = 1$	$w_s = 10^{12}, w_c = 10^5$
run [2900]: Style Loss : 639924.187500 Content Loss: 3389178.000000	run [2900]: Style Loss : 1067928128.000000 Content Loss: 9557970.000000	run [2900]: Style Loss : 65174564864.000000 Content Loss: 336528375808.000000
		

위의 그림을 보면 가운데 그림이 style image에 나타난 물결 모양의 선이 가장 두드러지게 나타난 것으로 확인되었다. 가운데 그림에 비해 왼쪽 그림은 style weight가 작고 오른쪽 그림은 content weight가 크다. 따라서, 원본 이미지(content image)의 스타일을 최대한 두드러지게 나타내고 싶으면 style weight를 크게 설정하고, content weight를 작게 설정하면 된다(반대로 style weight가 작고 content weight가 크면 style image가 잘 반영이 안된다).

Step 4: Try with another style image

옛날에 아쿠아리움에 가서 찍었던 물고기 사진을 style image로, $w_s = 10^{12}, w_c = 1$ 로 설정하고 코드를 돌려보았다.

```
run [2900]:
Style Loss : 3574222336.000000 Content Loss: 7055002.000000
```



Style image의 전체적인 색깔이 초록색이기 때문에 output image도 초록빛깔로 나타났다. 그리고 수조 속 배경의 무늬들이 output image에도 반영이 되었다.