Resnet

2024.01.23

이은주

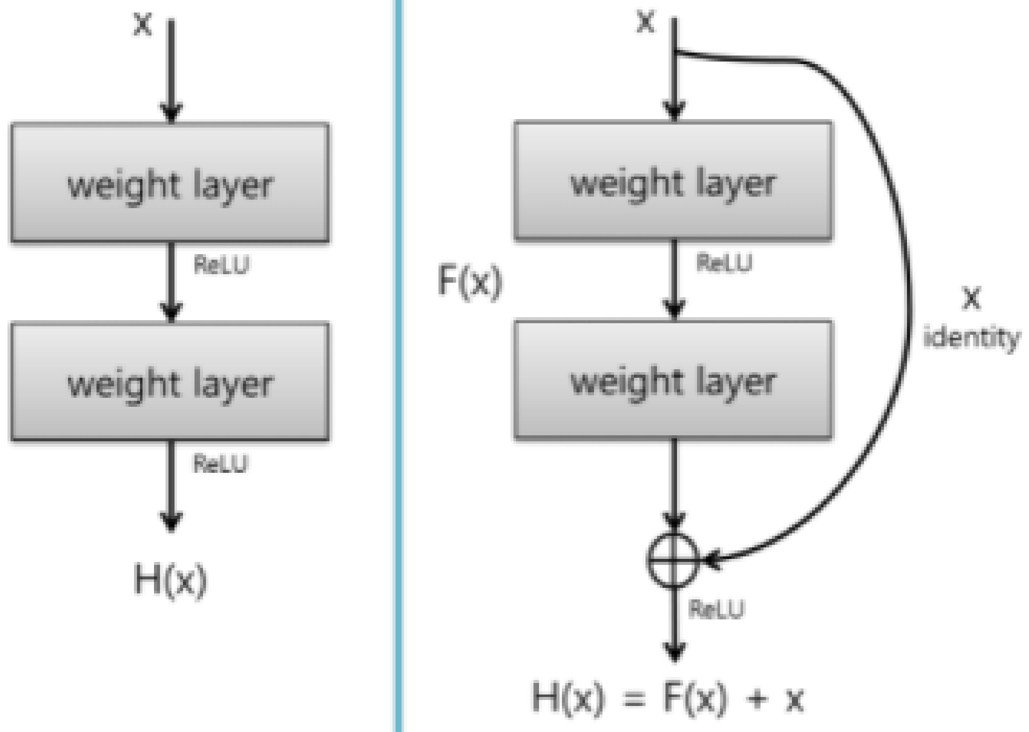
이전 CNN모델의 한계

- layer를 쌓는 것은 큰 성과를 쌓지 못함.

-> Gradient vanishing때문

- Resnet은 residual connection(skip connection) 개념을 사용하여 해결
- 네트워크를 deep하게 쌓을 수 있게 됨.

Residual Learning(skip connection)



기존 방식 Residual block

l 번째 있는 original block의 연산을 f(x)라 하고,
skip connection 의 연산을 h(x)라고 하자.

만약 h(x)=x로 identical 하다면,

$$x_{l+1} = x_l + f(x_l)$$

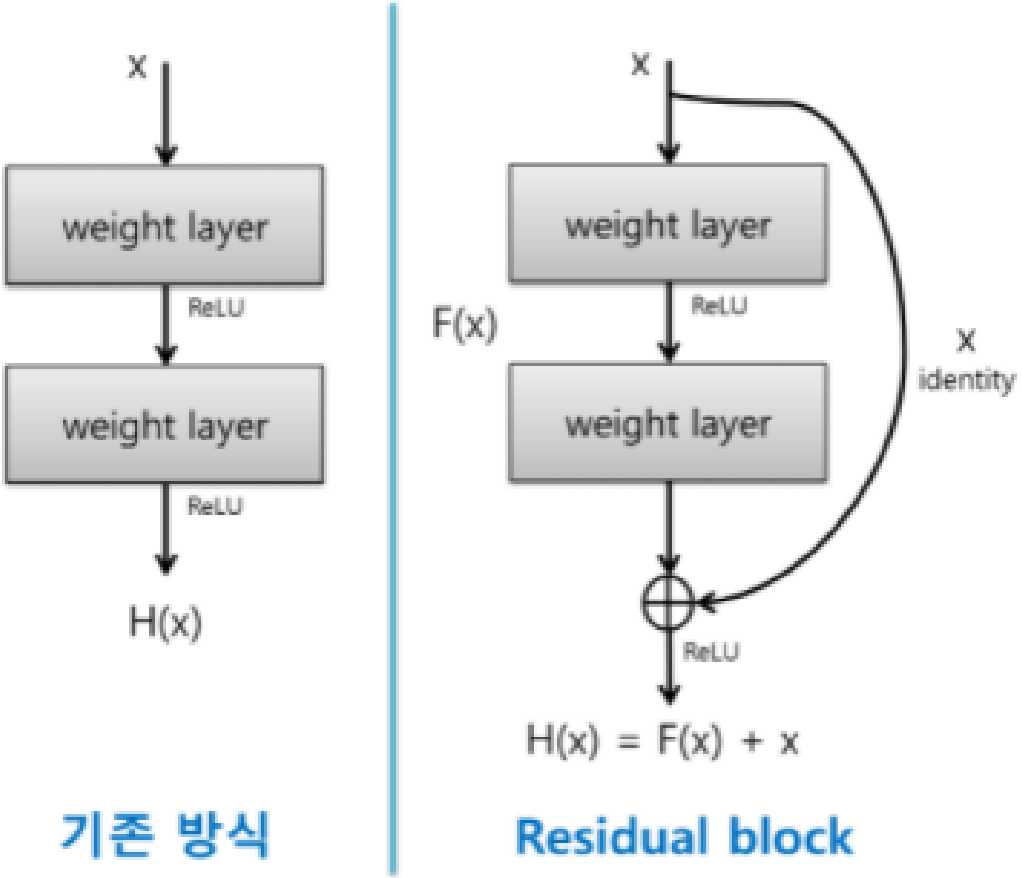
$$x_{l+2} = x_{l+1} + f(x_{l+1}) = x_l + f(x_1) + f(x_{l+1})$$

이고, 일반항을 구하면,

$$x_L(x) = x_l + \sum_{i=l}^{L-1} f(x_i)$$

를 만족한다. 즉, 여러번 쌓여도 x_l 은 남아있고,
Original block의 연산을 더할 뿐

Residual Learning(skip connection)



$$x_L(x) = x_l + \sum_{i=l}^{L-1} f(x_i)$$

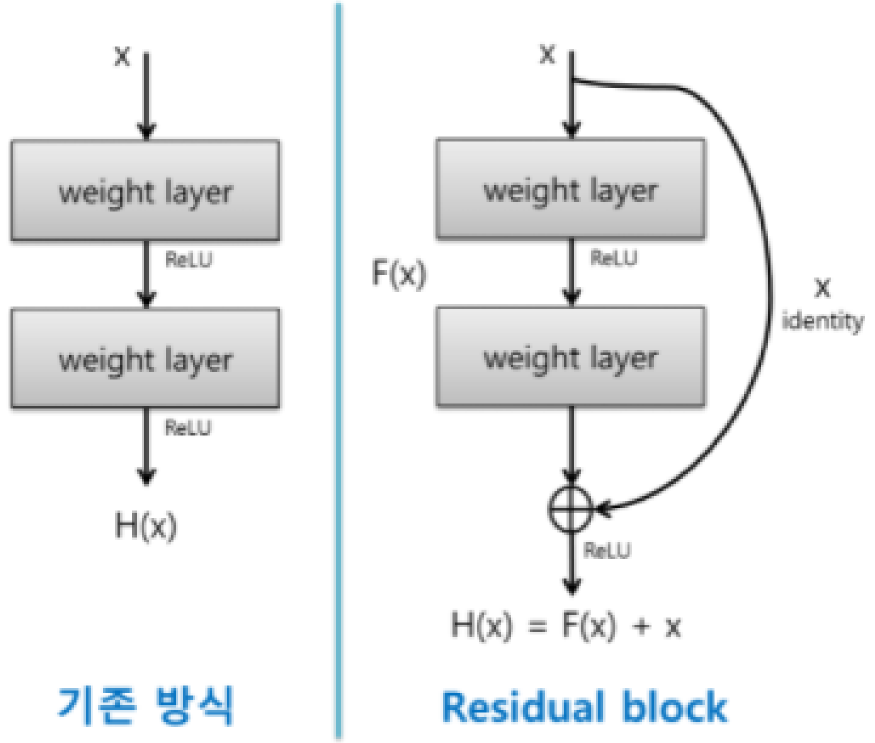
에서, backward propagation을 하여도,

$$\frac{\partial E}{\partial x_l} = \frac{\partial E}{\partial x_L}\frac{\partial x_L}{\partial x_l} = \frac{\partial E}{\partial x_L}\left(1 + \frac{\partial}{\partial x_l}\sum_{i}^{L} f(x_i)\right)$$

를 만족한다. 즉, 여러번 쌓여도 x_l 은 남아있고,
Original block의 gradient 연산을 더할 뿐

=> Gradient vanishing 해결하고 심지어 연산이 크게 추가 되지도 않는다.

Residual Learning




```python
# Resnet18 model
class BasicBlock(nn.Module):
    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        '''
        in_planes : 입력 필터개수
        out_planes : 출력 필터개수
        '''
        # 3x3 필터를 사용 (너비와 높이를 줄일 때는 stride값 조절)
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size = 3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes) # 배치정규화

        # 3x3 필터를 사용 (패딩1이므로 이미지가 동일하게 나옴)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, planes, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x) # 핵심 부분
        out = F.relu(out)
        return out
```

x : 입력값
F(x) : CNN Layer -> ReLU -> CNN Layer 을 통과한 출력값
H(x) : CNN Layer -> ReLU -> CNN Layer -> ReLU 를 통과한 출력값

nn.Sequential : nn.Linear, nn.ReLU(활성화 함수)같은 모듈들을 인수로 받아서 순서대로 정렬해놓고 입력값이 들어오면 순서대로 모듈을 실행해서 결과값을 리턴

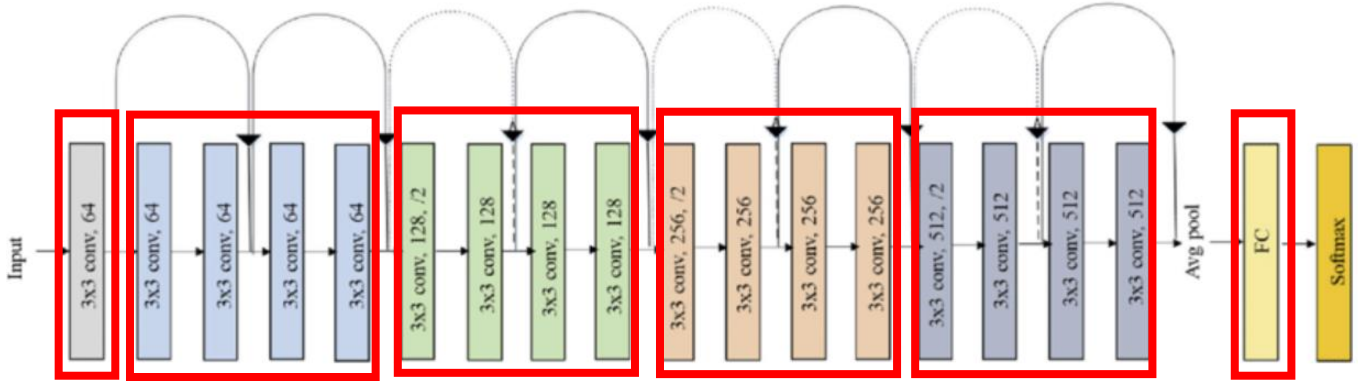
Resnet18

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		[3×3, 64 3×3, 64] ×2	[3×3, 64 3×3, 64] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3
conv3_x	28×28	[3×3, 128 3×3, 128] ×2	[3×3, 128 3×3, 128] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×8
conv4_x	14×14	[3×3, 256 3×3, 256] ×2	[3×3, 256 3×3, 256] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×23	[1×1, 256 3×3, 256 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512 3×3, 512] ×2	[3×3, 512 3×3, 512] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

ures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of block

```python
class Resnet(nn.Module):
    def __init__(self, block, num_blocks, num_class=10):
        super(Resnet, self).__init__()
        self.in_planes = 64

        # 64개의 3x3필터 사용
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        self.linear = nn.Linear(512, num_class)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out

def Resnet18():
    return Resnet(BasicBlock, [2,2,2,2])
```

Resnet18




```python
class Resnet(nn.Module):
    def __init__(self, block, num_blocks, num_class=10):
        super(Resnet, self).__init__()
        self.in_planes = 64

        # 64개의 3x3필터 사용
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        self.linear = nn.Linear(512, num_class)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes
        return nn.Sequential(*layers)
```

```python
    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out

def Resnet18():
    return Resnet(BasicBlock, [2,2,2,2])
```

- _make_layer호출 시 BasicBlock이 block으로 들어가므로 residual block(layer 2개)생성.
- num_blocks으로 basic block이 2개씩 들어가게됨.

Train

- Dataset : CIFAR10 dataset
- Train : 50000장
- Test : 10000장
- Image size : 32 x 32 x 3
- Batchsize : 128

- Train 실행 시 output
- Values : 확률 실수 값
- Indices : 클래스

- 이 중 가장 높은 확률의 클래스를 출력

```python
# train
def train(epoch):
    print(f"\n[ Train epoch : {epoch}]")
    net.train() # 모델을 학습모드로 설정
    train_loss = 0
    correct = 0
    total = 0

    for batch_idx, (inputs, targets) in enumerate(train_loader):
        inputs, targets = inputs.to(device), targets.to(device) # image와 target을 장비에 할당
        optimizer.zero_grad() #optimizer gradient 초기화

        outputs = net(inputs) #장비에 할당된 이미지를 모델의 input으로 이용해 output을 계산
        #print("output : ", outputs)
        #print("output.shape : ", outputs.shape) #output.shape :  torch.Size([128, 10]) 128개 이미지 각각 class 10에 대한 확률
        loss = criterion(outputs, targets) # 계산된 output과 target을 criterion(CrossEntropy)를 이용해서 loss 계산
        #print("loss : ", loss) # loss :  tensor(2.38660)
        loss.backward() # loss 계산한 결과를 바탕으로 back propagation을 통해 계산된 gradient값을 각 파라미터에 할당

        optimizer.step() # gradient값을 이용해 파라미터값 업데이트
        train_loss += loss.item() # tensor에 하나의 값만 존재한다면 scalar값을 얻을 수 있음. 만일 여러개 존재한다면 사용 불가.

        '''
        for i in range(outputs.size(1)): #10
            print(outputs[i])'''
        # outputs[0] : 첫번째 이미지에 대한 10개의 클래스 중 확률 값. 첫번째 이미지의 label은 9.
        # tensor([-0.6967,  0.4949, -0.3854,  0.6380,  0.4872, -0.4960, -1.0212,  0.2237, 0.5431, -0.8949], device='cuda:0',
        grad_fn=<SelectBackward0>)
        '''
        for j in range(len(outputs.max(1))):
            print("output max :", outputs.max(1))'''
        _, predicted = outputs.max(1) # output의 크기가 배치크기x클래스개수. 최댓값과 최댓값의 위치를 산출. _으로 처리하여 해당 출력값은
        저장하지 않고, 최댓값의 위치만 predicted에 저장하겠다.
        # values=tensor([3.3213, 1.3654, 3.1423, 2.0251, 1.9749, 2.1859, 1.1904, 2.1445, 2.3648, ... ] 128개
        # indices=tensor([5, 9, 1, 1, 5, 8, 9, 5, 7, 5, 0, 8, 7, 1, 9, 5, 4, 1, 1, 5, 1, 5, 6, 2, ... ] => predicted. 128개

        total += targets.size(0) # 128
        current_correct = predicted.eq(targets).sum().item() # 배열과 targets가 일치하는지 검사하고 sum으로 일치하는 것들의 개수의 합을
        숫자로 출력
        correct += current_correct

        if batch_idx % 100 == 0:
            print('\nCurrent batch : ', str(batch_idx))
            print(f'Current batch average train accuracy : {current_correct}/{targets.size(0)} => {current_correct/targets.size(0)}')
            print(f'Current batch average train loss : {loss.item()}/{targets.size(0)} => {loss.item()/targets.size(0)}')

    print(f'\nTotal average train accuracy : {correct}/{total} => {correct/total}')
    print(f'Total average train loss : {train_loss}/{total} => {train_loss/total}')
```

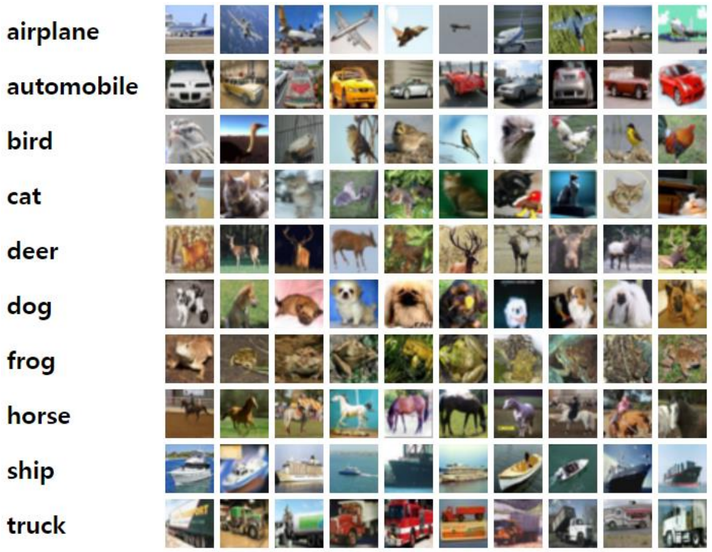
Test

```python
# test
def test(epoch):
    print('\n[Test epoch : %d]' % epoch)
    net.eval()
    loss = 0
    correct = 0
    total = 0
    
    for batch_idx, (inputs, targets) in enumerate(test_loader):
        inputs, targets = inputs.to(device), targets.to(device)
        total += targets.size(0) # 128
        
        outputs = net(inputs)
        loss += criterion(outputs, targets).item()
        
        _, predicted = outputs.max(1)
        correct += predicted.eq(targets).sum().item()
        
    print('\nTotal average test accuracy : ', correct/total)
    print('Total average test loss : ', loss/total)
    
    state = {
        'net' : net.state_dict()
    }
    if not os.path.isdir('checkpoint'):
        os.mkdir('checkpoint')
    torch.save(state, './checkpoint/' + file_name)
    print('Model Saved!')
    


def adjust_learning_rate(optimizer, epoch):
    lr = learning_rate
    if epoch >= 50:
        lr /=10
    if epoch >= 100:
        lr /= 10
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
```

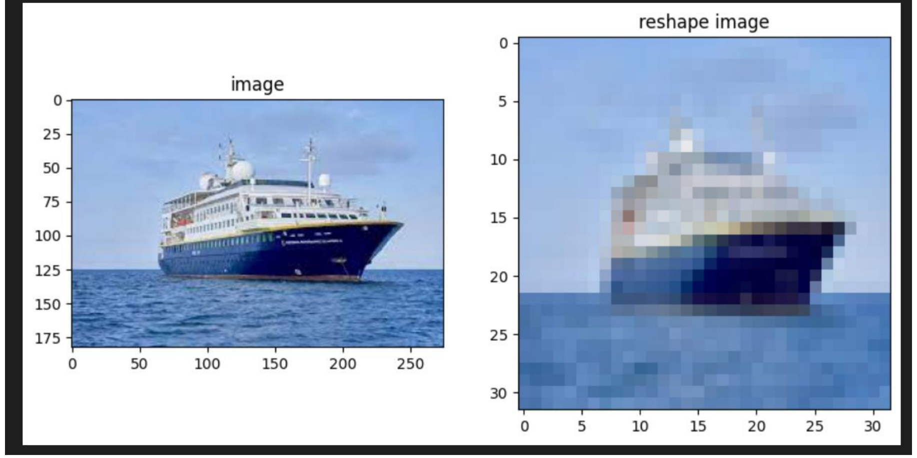
```python
import time

def adjust_learning_rate(optimizer, epoch):
    lr = learning_rate
    if epoch >= 50:
        lr /=10
    if epoch >= 100:
        lr /= 10
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
        
start_time = time.time()

for epoch in range(0,50):
    adjust_learning_rate(optimizer, epoch)
    train(epoch)
    test(epoch)
    print('\nTime elapsed : ', time.time() - start_time)
```

```
[ Train epoch : 0]

Current batch :  0
Current batch average train accuracy : 9/128 => 0.0703125
Current batch average train loss : 2.422952890396118/128 => 0.018929319456219673

Current batch :  100
Current batch average train accuracy : 41/128 => 0.3203125
Current batch average train loss : 1.8995040655136108/128 => 0.014839875511825085

Current batch :  200
Current batch average train accuracy : 46/128 => 0.359375
Current batch average train loss : 1.6757712364196777/128 => 0.013091962784528732

Current batch :  300
Current batch average train accuracy : 45/128 => 0.3515625
Current batch average train loss : 1.7577013969421387/128 => 0.013732042163610458

Total average train accuracy : 15120/50000 => 0.3024
Total average train loss : 751.9228405952454/50000 => 0.015038456811904907

[Test epoch : 0]

Total average test accuracy :  0.4013
...
Total average test loss :  0.0028937289983034134
Model Saved!
```

새로운 Data 예측(pretrained)




```
(275, 183)
(32, 32)
```




```python
# pretrain 준비
torch.load('./checkpoint/resnet18_cifar10.pth')
```

```
net': OrderedDict([('conv1.weight',
              tensor([[[[ 1.9007e-02,  7.0236e-02,  7.6553e-02],
                        [ 3.6165e-02,  9.3259e-02,  9.6969e-02],
                        [-2.0013e-02,  1.0872e-02,  5.7526e-02]],

                       [[-2.6176e-02,  4.3282e-02,  5.6957e-02],
                        [-2.3329e-02,  3.2779e-02,  6.1962e-02],
                        [-5.4937e-02, -4.2050e-02,  9.0840e-03]],

                       [[-4.1336e-02,  1.8785e-02,  3.5406e-02],
                        [-5.0364e-02, -1.3069e-02,  6.8028e-03],
                        [-7.8665e-02, -8.8680e-02, -4.3480e-02]]],


                      [[[ 6.1746e-02, -2.7466e-02, -1.1448e-01],
                        [-3.0563e-02, -4.4555e-02,  6.4489e-02],
                        [ 6.0118e-03, -5.7901e-02,  4.4292e-02]],

                       [[ 8.0762e-02,  2.2446e-02, -3.5991e-02],
                        [-6.1249e-02, -6.8931e-02,  8.6970e-02],
                        [-3.7146e-02, -1.1523e-01,  2.8131e-02]],

                       [[ 6.8880e-02,  3.5894e-02, -2.8244e-02],
                        [-2.4070e-02, -1.4089e-02,  9.1884e-02],
                        [-2.8364e-02, -7.3092e-02,  4.5346e-02]]],

                      [-0.1250, -0.0791,  0.2862,  ..., -0.1580, -0.1440,  0.2127]],
                     device='cuda:0')),
             ('linear.bias',
              tensor([ 0.1025, -0.2748,  0.1330,  0.2847,  0.1084,  0.1228, -0.0834, -0.0844,
                      -0.0980, -0.2110], device='cuda:0'))])}
```

```python
# 예측된 클래스 확인
print("예측된 클래스:", predicted_class)
```

```
예측된 클래스: 8
```