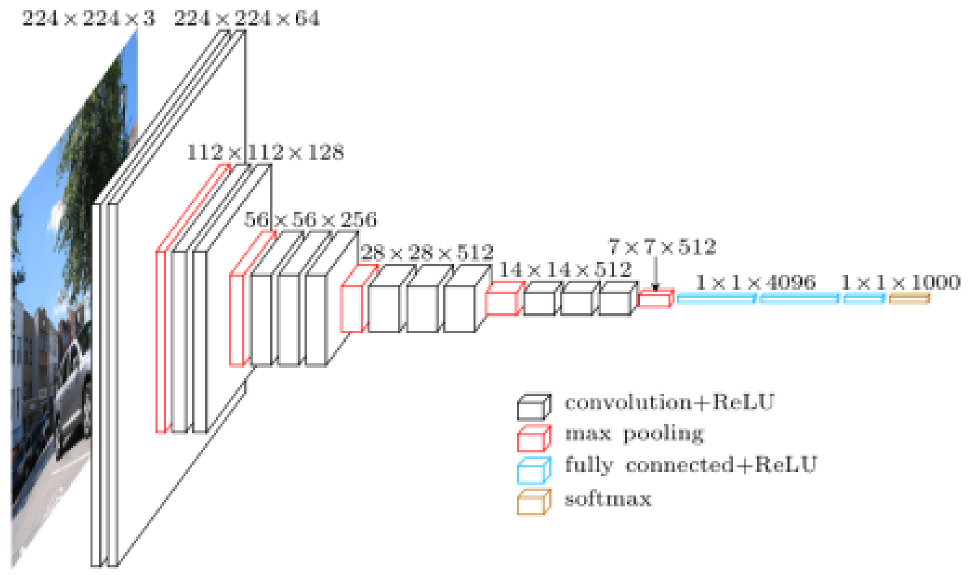
Dense With Me

-Densely Connected Convolutional Networks-

Dense Connectivity?



기존의 CNN구조. 커널의 차원에 맞춰 채널이 변경되며,
그 과정에서 정보의 소실이 발생하고, 깊어질수록 곱연산으로 파라미터 급증.

Dense Connectivity!

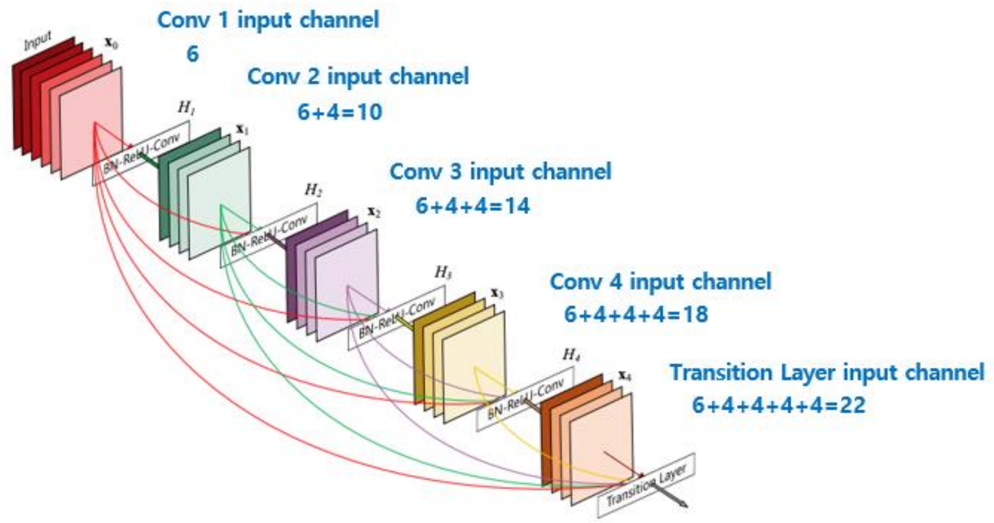


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

6,6+4,6+4+4,.....Feature map이 channel-wise로 Growth rate 인 4씩 쌓이는 중.

각자의 Feature map이 채널이 급증하는 것을 막기 위해 더해지는
채널의 숫자는 작게 설정합니다 (Growth rate)
자세히 보시면 Feature map의 spatial dimension은 변하지 않습니다.
모든 Feature가 손상 없이 쌓이는 중이라는 뜻입니다.

Dense Connectivity!

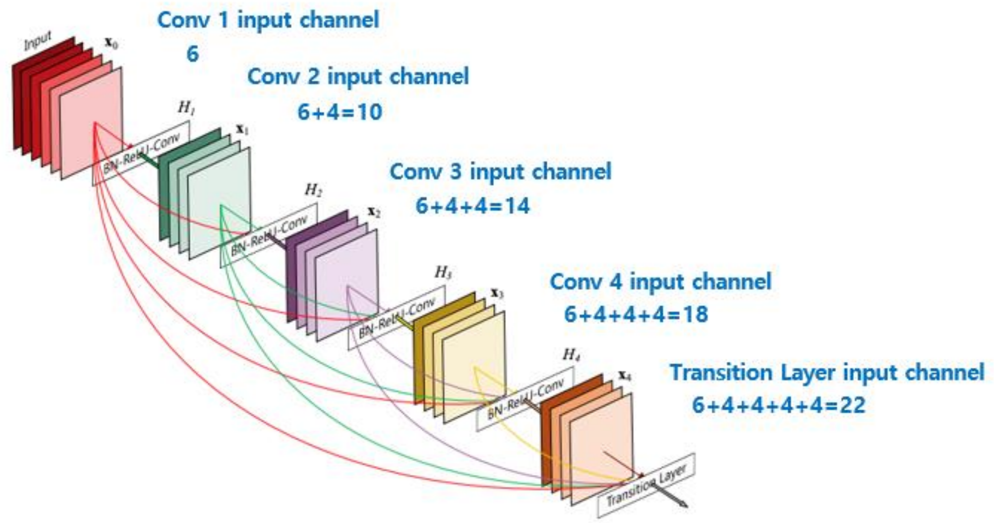
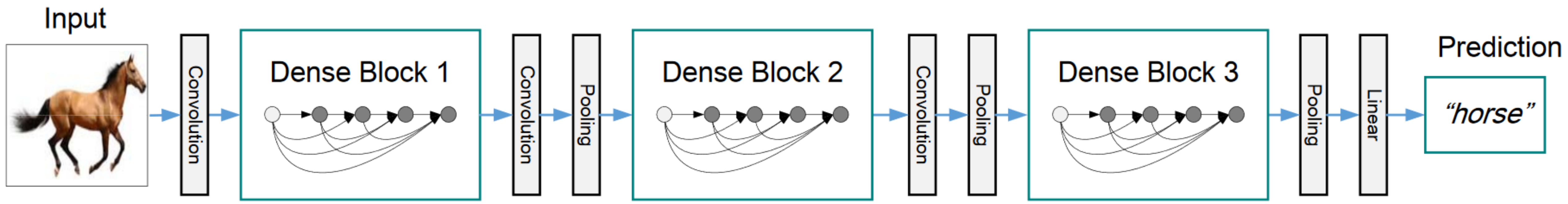


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

그래디언트 소실 문제 개선, 특성 전달 강화(특성 재사용)
파라미터 수 절감.

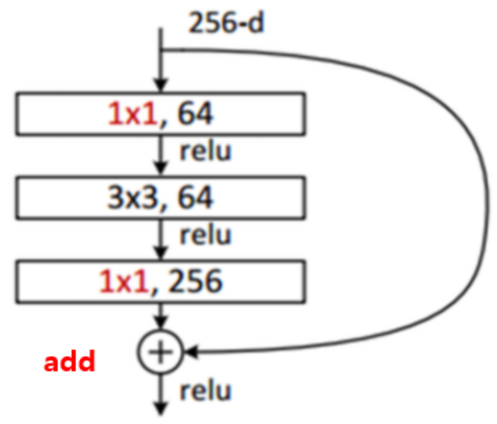
Whole Network



Dense Connectivity!

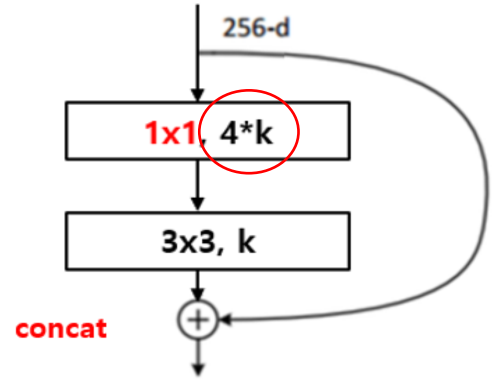
$$\mathbf{x}_\ell = H_\ell(\mathbf{x}_{\ell-1}) + \mathbf{x}_{\ell-1}.$$

$$\mathbf{x}_\ell = H_\ell([\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{\ell-1}]),$$



bottleneck

(for ResNet)



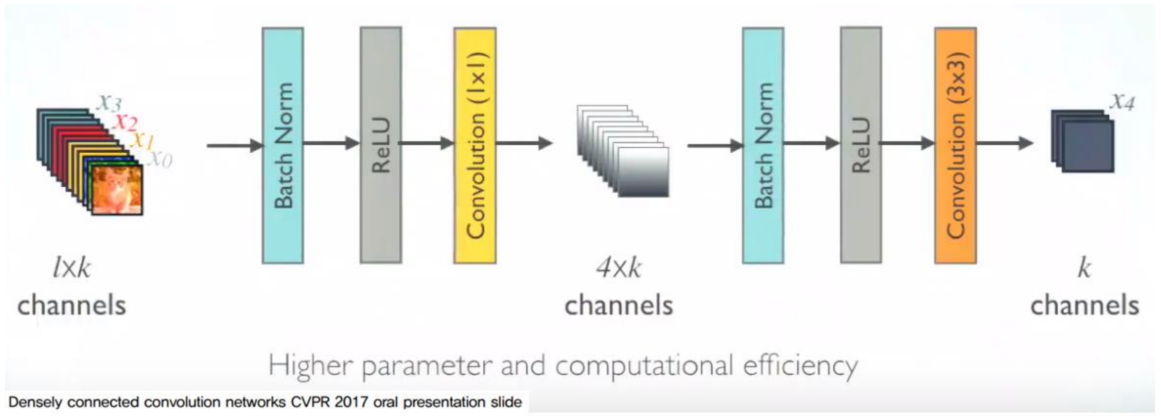
4*growth rate 라는 수치는 별도의 설명이 없습니다.
이게 그냥 제일 잘되나 봐요.

bottleneck

(for DenseNet)

Bottleneck layer에서 채널의 차원을 축소할 때.
ResNet은 x항을 통해 그래디언트를 전달하나, Add하는 방식이기에 정보 소실 발생 가능.
이번 레이어에서 지켜진 정보가, 다음 레이어에서 잔차로서 날아갈 가능성 있음.
DenseNet은 Add가 아닌 Concat. 다음 레이어에서도 정보는 보존됨. 정보의 구분이 가능.

Composite Function



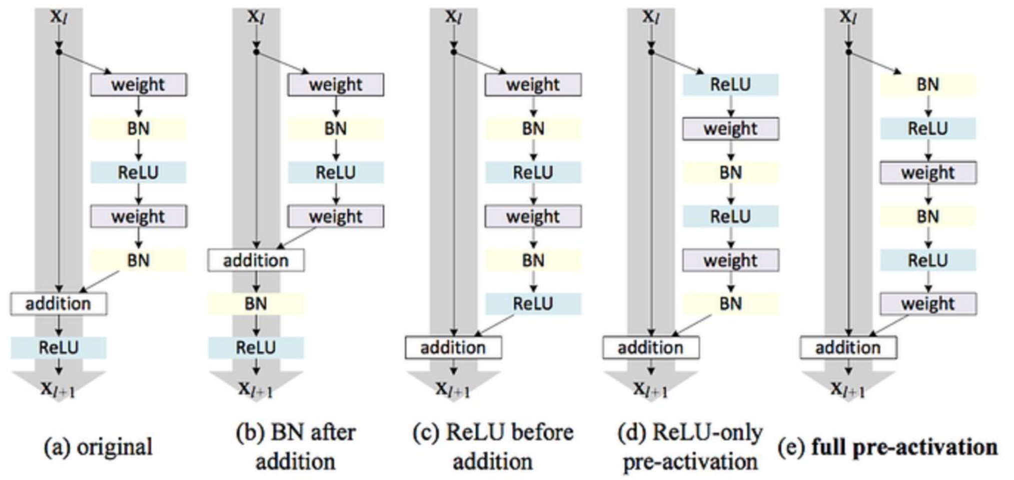
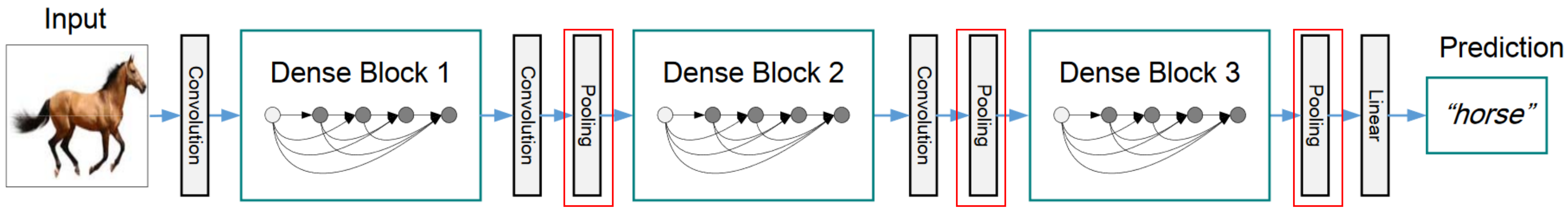


Figure 4. Various usages of activation in Table 2. All these units consist of the same components — only the orders are different.

"Identity mappings in deep residual networks, 2016

Dense block 에서 쌓여버린 feature map으로 높아진 계산 복잡성을
BN_RELU_CONV를 통해 줄여준다.
BN_RELU_CONV가 반복되기에, 따로 구현해놓으면 좋다.

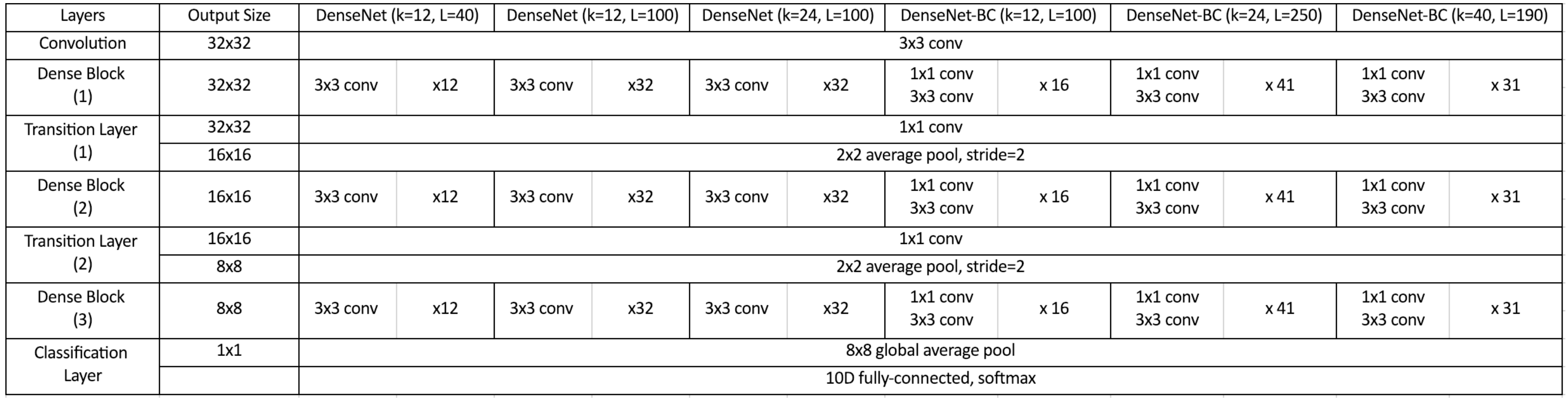
Transition Layer



Transition Layer.

일종의 풀링층으로, 특성 맵 차원 축소를 위한 레이어.
1x1 convolution으로 특성맵을 줄여주며, 이때의 정도를 theta라는 하이퍼파라미터를 적용.
이 과정을 논문에서는 Compression이라고 부름.
또한 average pooling layer가 포함되어, 가로 세로 크기 또한 평균인 절반으로 줄어듬.

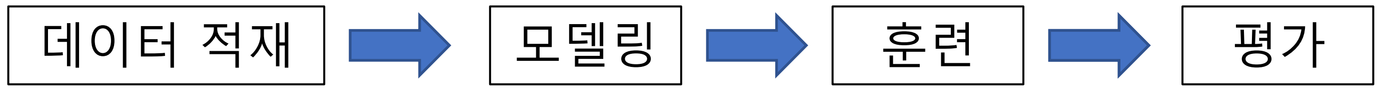
Architecture



Layers	Output Size	DenseNet (k=12, L=40)		DenseNet (k=12, L=100)		DenseNet (k=24, L=100)		DenseNet-BC (k=12, L=100)		DenseNet-BC (k=24, L=250)		DenseNet-BC (k=40, L=190)	
Convolution	32x32	3x3 conv											
Dense Block (1)	32x32	3x3 conv	x12	3x3 conv	x32	3x3 conv	x32	1x1 conv 3x3 conv	x 16	1x1 conv 3x3 conv	x 41	1x1 conv 3x3 conv	x 31
Transition Layer (1)	32x32	1x1 conv											
	16x16	2x2 average pool, stride=2											
Dense Block (2)	16x16	3x3 conv	x12	3x3 conv	x32	3x3 conv	x32	1x1 conv 3x3 conv	x 16	1x1 conv 3x3 conv	x 41	1x1 conv 3x3 conv	x 31
Transition Layer (2)	16x16	1x1 conv											
	8x8	2x2 average pool, stride=2											
Dense Block (3)	8x8	3x3 conv	x12	3x3 conv	x32	3x3 conv	x32	1x1 conv 3x3 conv	x 16	1x1 conv 3x3 conv	x 41	1x1 conv 3x3 conv	x 31
Classification Layer	1x1	8x8 global average pool											
		10D fully-connected, softmax											

눈에 잘 안들어오죠…? 코드로 보면 나오실 겁니다.

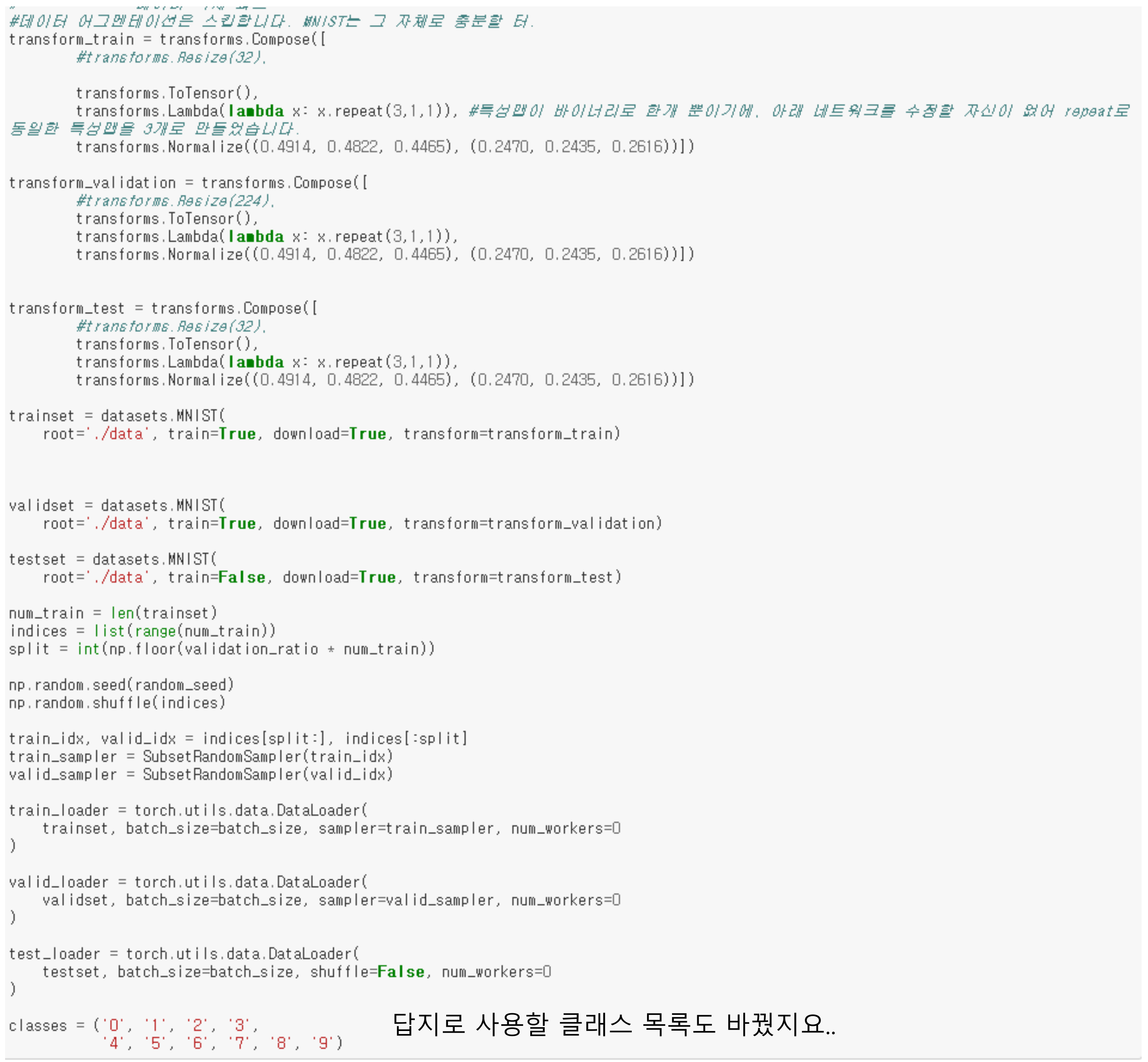
Implementation



Implementation, 데이터셋 처리.

CIFAR10을 위해 설계된 네트워크, MNIST는 특성맵 차원 부족.
전처리 시 복제했습니다.

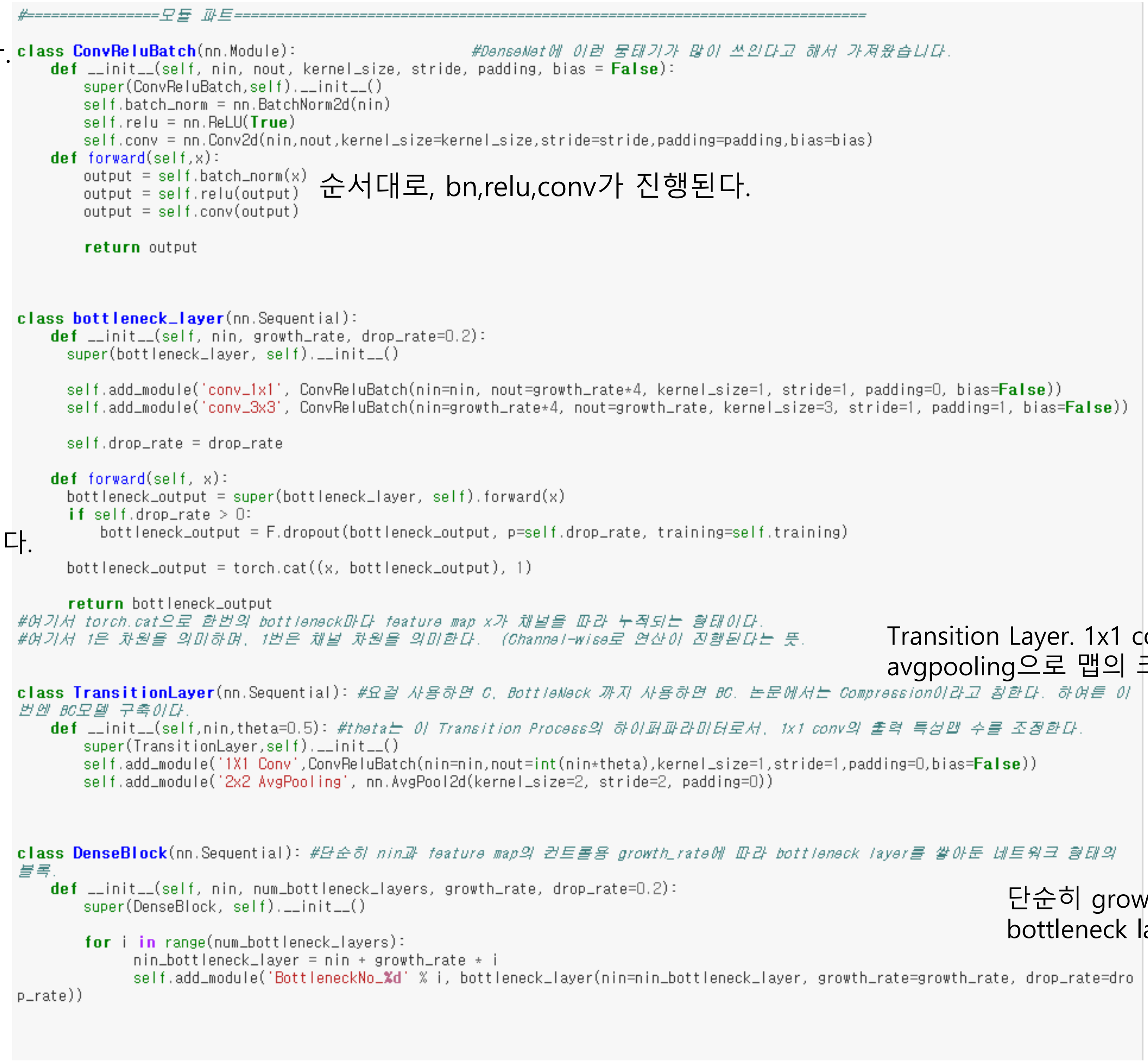
원본 코드에서 데이터셋만 MNIST로 바꿨습니다.



```
#데이터 어그멘테이션은 스킵합니다. MNIST는 그 자체로 충분할 터.
transform_train = transforms.Compose([
        #transforms.Resize(32),

        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.repeat(3,1,1)), #특성맵이 바이너리로 한개 뿐이기에. 아래 네트워크를 수정할 자신이 없어 repeat로
동일한 특성맵을 3개로 만들었습니다.
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))])

transform_validation = transforms.Compose([
        #transforms.Resize(224),
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.repeat(3,1,1)),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))])


transform_test = transforms.Compose([
        #transforms.Resize(32),
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.repeat(3,1,1)),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))])

trainset = datasets.MNIST(
    root='./data', train=True, download=True, transform=transform_train)



validset = datasets.MNIST(
    root='./data', train=True, download=True, transform=transform_validation)

testset = datasets.MNIST(
    root='./data', train=False, download=True, transform=transform_test)

num_train = len(trainset)
indices = list(range(num_train))
split = int(np.floor(validation_ratio * num_train))

np.random.seed(random_seed)
np.random.shuffle(indices)

train_idx, valid_idx = indices[split:], indices[:split]
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = torch.utils.data.DataLoader(
    trainset, batch_size=batch_size, sampler=train_sampler, num_workers=0
)

valid_loader = torch.utils.data.DataLoader(
    validset, batch_size=batch_size, sampler=valid_sampler, num_workers=0
)

test_loader = torch.utils.data.DataLoader(
    testset, batch_size=batch_size, shuffle=False, num_workers=0
)

classes = ('0', '1', '2', '3',
           '4', '5', '6', '7', '8', '9')
```

답지로 사용할 클래스 목록도 바꿨지요..

Implementation,모델링-1



클래스 명은 실수였다.

```
#===============모듈 파트=================================================================================

class ConvReluBatch(nn.Module):                    #DenseNet에 이런 몸태기가 많이 쓰인다고 해서 가져왔습니다.
    def __init__(self, nin, nout, kernel_size, stride, padding, bias = False):
        super(ConvReluBatch,self).__init__()
        self.batch_norm = nn.BatchNorm2d(nin)
        self.relu = nn.ReLU(True)
        self.conv = nn.Conv2d(nin,nout,kernel_size=kernel_size,stride=stride,padding=padding,bias=bias)
    def forward(self,x):
        output = self.batch_norm(x)
        output = self.relu(output)
        output = self.conv(output)

        return output
```

순서대로, bn,relu,conv가 진행된다.

앞 슬라이드의 병목레이어

```
class bottleneck_layer(nn.Sequential):
    def __init__(self, nin, growth_rate, drop_rate=0.2):
      super(bottleneck_layer, self).__init__()

      self.add_module('conv_1x1', ConvReluBatch(nin=nin, nout=growth_rate*4, kernel_size=1, stride=1, padding=0, bias=False))
      self.add_module('conv_3x3', ConvReluBatch(nin=growth_rate*4, nout=growth_rate, kernel_size=3, stride=1, padding=1, bias=False))

      self.drop_rate = drop_rate

    def forward(self, x):
      bottleneck_output = super(bottleneck_layer, self).forward(x)
      if self.drop_rate > 0:
          bottleneck_output = F.dropout(bottleneck_output, p=self.drop_rate, training=self.training)

      bottleneck_output = torch.cat((x, bottleneck_output), 1)

      return bottleneck_output
#여기서 torch.cat으로 한번의 bottleneck마다 feature map x가 채널을 따라 누적되는 형태이다.
#여기서 1은 차원을 의미하며. 1번은 채널 차원을 의미한다.  (Channel-wise로 연산이 진행된다는 뜻.
```

Torch.cat = concat함수이다.

Transition Layer. 1x1 conv로 차원을 줄이고, avgpooling으로 맵의 크기를 줄였다.

```
class TransitionLayer(nn.Sequential): #요걸 사용하면 C. BottleNeck 까지 사용하면 BC. 논문에서는 Compression이라고 칭한다. 하여튼 이
번엔 BC모델 구축이다.
    def __init__(self,nin,theta=0.5): #theta는 이 Transition Process의 하이퍼파라미터로서. 1x1 conv의 출력 특성맵 수를 조정한다.
        super(TransitionLayer,self).__init__()
        self.add_module('1X1 Conv',ConvReluBatch(nin=nin,nout=int(nin*theta),kernel_size=1,stride=1,padding=0,bias=False))
        self.add_module('2x2 AvgPooling', nn.AvgPool2d(kernel_size=2, stride=2, padding=0))


class DenseBlock(nn.Sequential): #단순히 nin과 feature map의 컨트롤용 growth_rate에 따라 bottleneck layer를 쌓아둔 네트워크 형태의
블록.
    def __init__(self, nin, num_bottleneck_layers, growth_rate, drop_rate=0.2):
        super(DenseBlock, self).__init__()

        for i in range(num_bottleneck_layers):
            nin_bottleneck_layer = nin + growth_rate * i
            self.add_module('BottleneckNo_%d' % i, bottleneck_layer(nin=nin_bottleneck_layer, growth_rate=growth_rate, drop_rate=dro
p_rate))
```

단순히 growth rate에 따라 bottleneck layer를 쌓은 블록.

Implementation,모델링-2

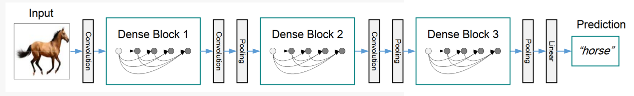
앞에서 본 아키텍처에 따라
레이어를 쌓아준다.

주석에 달린 차원의 변화를 주목해주세요

완성된 네트워크는 forward를
보시면 됩니다.

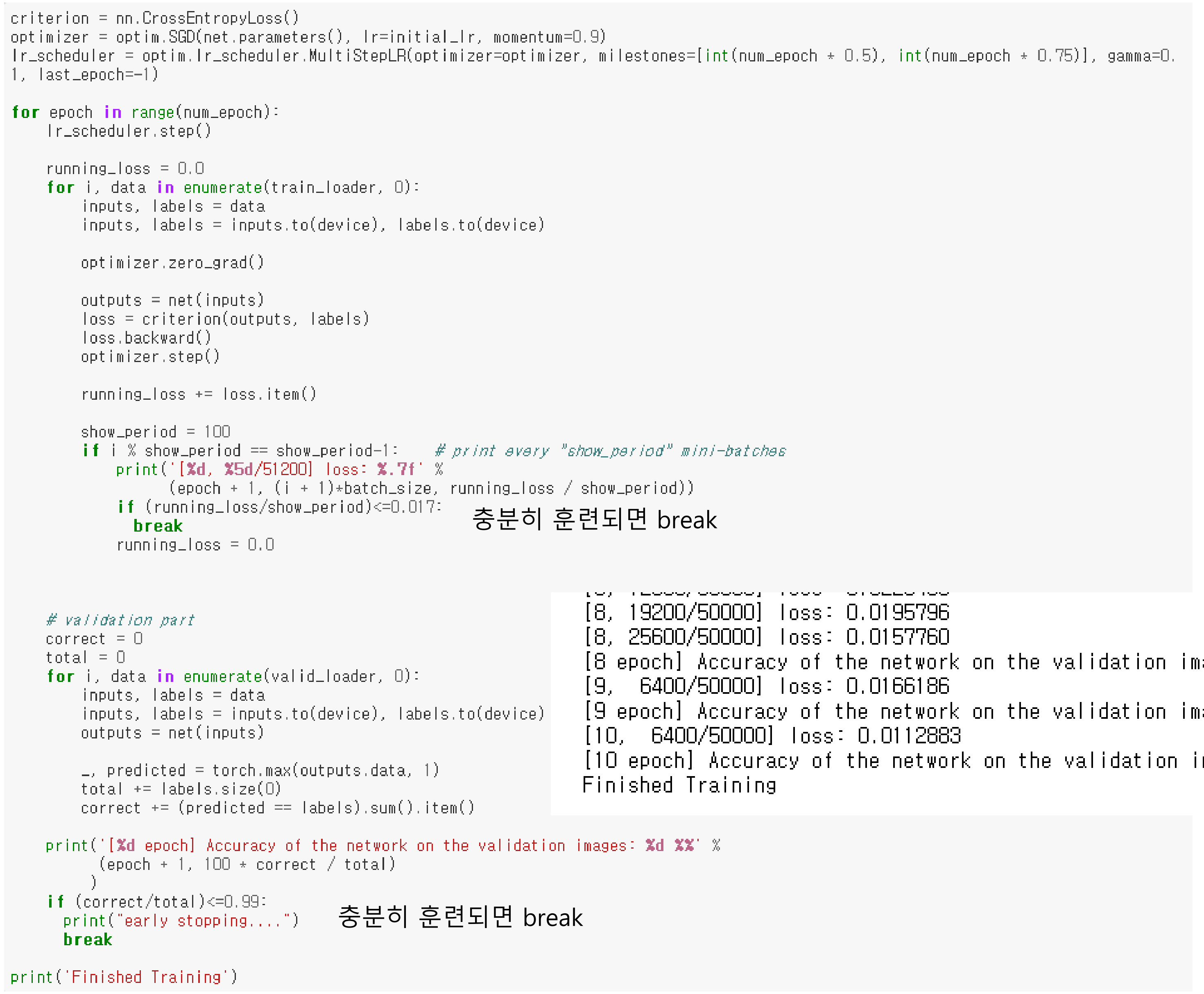
```python
class DenseNet(nn.Module):
    def __init__(self, growth_rate =12 , num_layers=100,theta=0.5,drop_rate=0.2,num_classes=10):
        super(DenseNet,self).__init__()
        assert ( num_layers - 4)%6 == 0

        num_bottleneck_layers=(num_layers-4)//6

        self.dense_init = nn.Conv2d(3,growth_rate*2,kernel_size=3,stride=1,padding=1,bias=True)
       # 32 x 32 x (growth_rate*2) --> 32 x 32 x [(growth_rate*2) + (growth_rate * num_bottleneck_layers)]
        self.dense_block_1 = DenseBlock(nin=growth_rate*2, num_bottleneck_layers=num_bottleneck_layers, growth_rate=growth_rate, drop_
rate=drop_rate)

        # 32 x 32 x [(growth_rate*2) + (growth_rate * num_bottleneck_layers)] --> 16 x 16 x [(growth_rate*2) + (growth_rate * num_bott
leneck_layers)]*theta
        nin_transition_layer_1 = (growth_rate*2) + (growth_rate * num_bottleneck_layers)
        self.transition_layer_1 = TransitionLayer(nin=nin_transition_layer_1, theta=theta)

        # 16 x 16 x nin_transition_layer_1*theta --> 16 x 16 x [nin_transition_layer_1*theta + (growth_rate * num_bottleneck_layers)]
        self.dense_block_2 = DenseBlock(nin=int(nin_transition_layer_1*theta), num_bottleneck_layers=num_bottleneck_layers, growth_rat
e=growth_rate, drop_rate=drop_rate)

        # 16 x 16 x [nin_transition_layer_1*theta + (growth_rate * num_bottleneck_layers)] --> 8 x 8 x [nin_transition_layer_1*theta +
(growth_rate * num_bottleneck_layers)]*theta
        nin_transition_layer_2 = int(nin_transition_layer_1*theta) + (growth_rate * num_bottleneck_layers)
        self.transition_layer_2 = TransitionLayer(nin=nin_transition_layer_2, theta=theta)

        # 8 x 8 x nin_transition_layer_2*theta --> 8 x 8 x [nin_transition_layer_2*theta + (growth_rate * num_bottleneck_layers)]
        self.dense_block_3 = DenseBlock(nin=int(nin_transition_layer_2*theta), num_bottleneck_layers=num_bottleneck_layers, growth_rat
e=growth_rate, drop_rate=drop_rate)

        nin_fc_layer = int(nin_transition_layer_2*theta) + (growth_rate * num_bottleneck_layers)

        # [nin_transition_layer_2*theta + (growth_rate * num_bottleneck_layers)] --> num_classes
        self.fc_layer = nn.Linear(nin_fc_layer, num_classes)

    def forward(self, x):
        dense_init_output = self.dense_init(x)

        dense_block_1_output = self.dense_block_1(dense_init_output)
        transition_layer_1_output = self.transition_layer_1(dense_block_1_output)

        dense_block_2_output = self.dense_block_2(transition_layer_1_output)
        transition_layer_2_output = self.transition_layer_2(dense_block_2_output)

        dense_block_3_output = self.dense_block_3(transition_layer_2_output)

        global_avg_pool_output = F.adaptive_avg_pool2d(dense_block_3_output, (1, 1))
        global_avg_pool_output_flat = global_avg_pool_output.view(global_avg_pool_output.size(0), -1)

        output = self.fc_layer(global_avg_pool_output_flat)

        return output

#각 DenseBlock 마다 같은 개수의 convolution 연산을 사용한다.


def DenseNetBC_100_12():
    return DenseNet(growth_rate=12, num_layers=100, theta=0.5, drop_rate=0.2, num_classes=10)
```

아까 본 그거



Implementation, 훈련

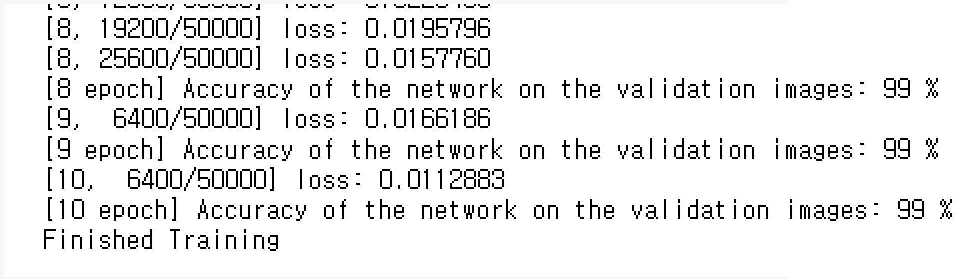
손실함수, optimizer,
학습률 scheduler



```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=initial_lr, momentum=0.9)
lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer=optimizer, milestones=[int(num_epoch * 0.5), int(num_epoch * 0.75)], gamma=0.
1, last_epoch=-1)

for epoch in range(num_epoch):
    lr_scheduler.step()

    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        show_period = 100
        if i % show_period == show_period-1:    # print every "show_period" mini-batches
            print('[%d, %5d/51200] loss: %.7f' %
                  (epoch + 1, (i + 1)*batch_size, running_loss / show_period))
            if (running_loss/show_period)<=0.017:
              break
            running_loss = 0.0


    # validation part
    correct = 0
    total = 0
    for i, data in enumerate(valid_loader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = net(inputs)

        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('[%d epoch] Accuracy of the network on the validation images: %d %%' %
          (epoch + 1, 100 * correct / total)
         )
    if (correct/total)<=0.99:
      print("early stopping....")
      break

print('Finished Training')
```

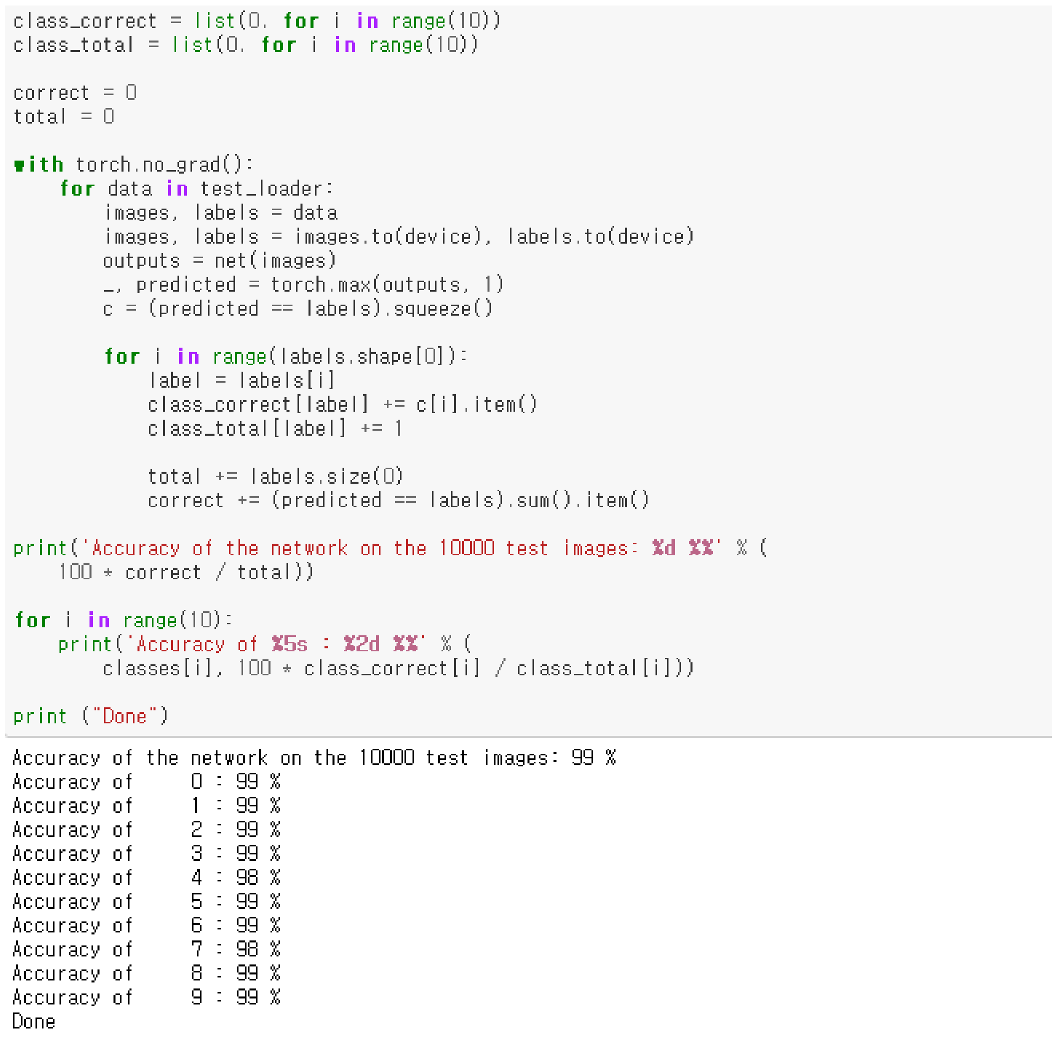
충분히 훈련되면 break

충분히 훈련되면 break



```
[8, 19200/50000] loss: 0.0195796
[8, 25600/50000] loss: 0.0157760
[8 epoch] Accuracy of the network on the validation images: 99 %
[9,  6400/50000] loss: 0.0166186
[9 epoch] Accuracy of the network on the validation images: 99 %
[10,  6400/50000] loss: 0.0112883
[10 epoch] Accuracy of the network on the validation images: 99 %
Finished Training
```

Implementation,평가



```python
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))

correct = 0
total = 0

with torch.no_grad():
    for data in test_loader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()

        for i in range(labels.shape[0]):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

            total += labels.size(0)
            correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))

for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))

print ("Done")
```

```
Accuracy of the network on the 10000 test images: 99 %
Accuracy of     0 : 99 %
Accuracy of     1 : 99 %
Accuracy of     2 : 99 %
Accuracy of     3 : 99 %
Accuracy of     4 : 98 %
Accuracy of     5 : 99 %
Accuracy of     6 : 99 %
Accuracy of     7 : 98 %
Accuracy of     8 : 99 %
Accuracy of     9 : 99 %
Done
```

Thank You for Listening!