7. Modern CNN

Junggwon Shin, shinjunggwon@gmail.com

Summary for Dive Into Deep Learning, https://d2l.ai/chapter_preface/index.html

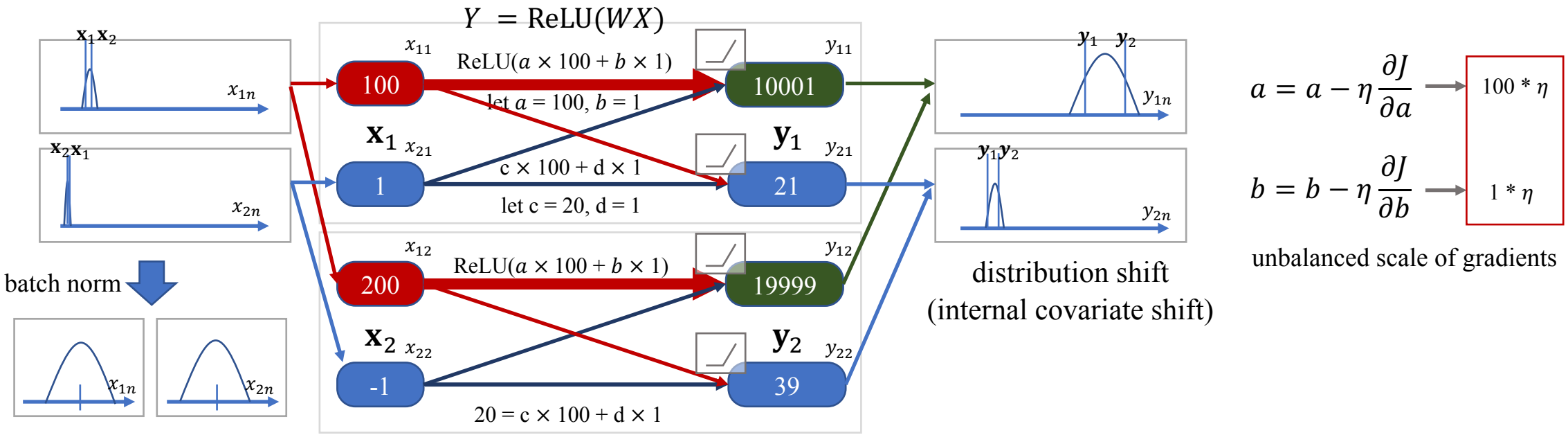
7.5 Batch Normalization

7.6 Residual Networks

7.7 Densely Connected Networks

Motivation

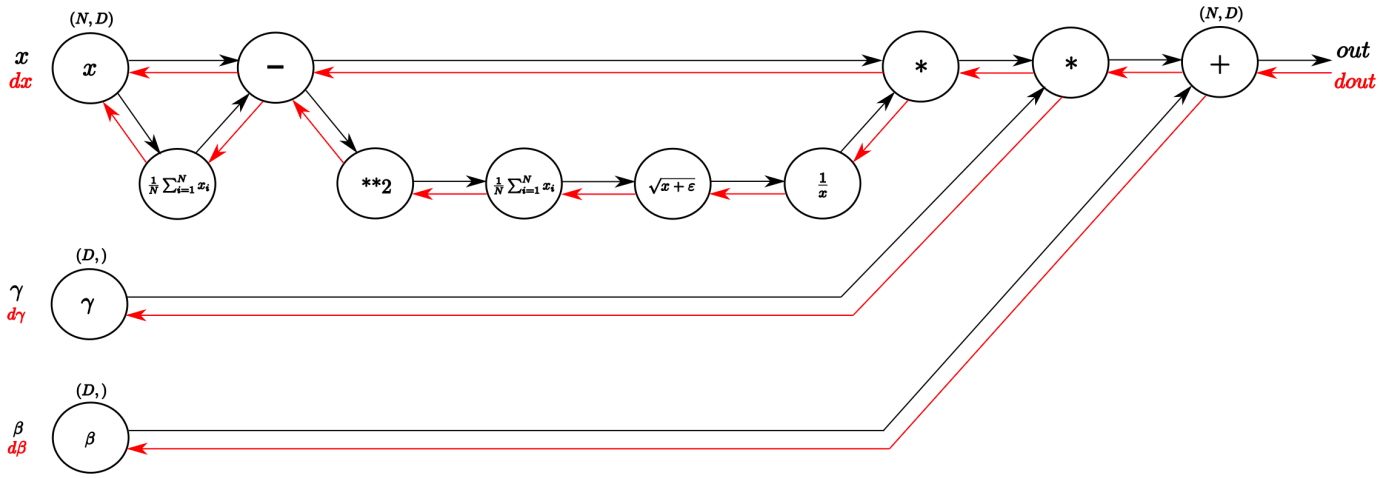
- Parameters in intermediate layers may take values with widely varying magnitudes due to learning and updates to the model parameters.
- This drift in the distribution of such parameters could hamper the convergence of the network due to their scale



- Standardization plays as *a priori* to make parameters a similar scale and accelerates optimizers for the convergence of deep networks
  - Normalize the current minibatch inputs by subtracting their mean and dividing by their standard deviation and applying a scale coefficient and a scale offset → **Reducing 'internal' covariate shift**
  - Not be able to learn anything for minibatches of size 1 → effective for **enough large minibatches** (50~100 range)

Formulation

- Batch normalization transforms input x $\in \mathcal{B}$ (minibatch) as



sample mean

$$\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}.$$

shift parameter

scale parameter

sample std

learning parameters

$$\hat{\boldsymbol{\mu}}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x},$$

$$\hat{\boldsymbol{\sigma}}^2_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon.$$

$\epsilon > 0$: Smoothing parameter to ensure that we never attempt division by zero

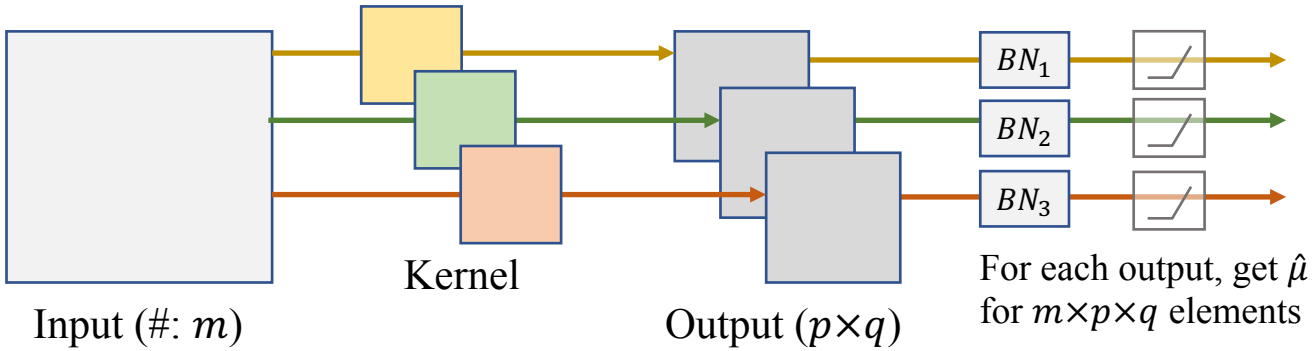
- Batch normalization behaves differently during training and at test time
- Training
  - Calculate $\hat{\mu}$ and $\hat{\sigma}$ for each minibatch
  - The estimates $\hat{\mu}$ and $\hat{\sigma}$ are different, varying, and noisy for each minibatch
    - The intermediate variables for all data examples change every time
  - Even though the reasons are not theoretically characterized, the noises in the estimates turn out to lead to faster learning and less overfitting by acting as a form of Bayesian priors and regularization.
- Prediction
  - Calculate the means and variances of each layer's variables based on the entire dataset and fix the values for prediction

Batch Norm in Fully-Connected Layers

- Applying batch norm before use of activation function $\mathbf{h} = \phi(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b}))$.

Batch Norm in CNN

- Applying batch norm after convolution or before use of activation and pooling
- Carry out each batch normalization over the $m \cdot p \cdot q$ elements (all elements) for each output channel




```
nn.Conv2D(channels=16, kernel_size=5, activation='sigmoid'),
nn.AvgPool2D(pool_size=2, strides=2),
```

```
nn.Conv2d(6, 16, kernel_size=5), nn.BatchNorm2d(16), nn.Sigmoid(),
nn.MaxPool2d(kernel_size=2, stride=2), nn.Flatten(),
```

Batch Normalization in LeNet

$batch \times 6 \times p \times q$

```
net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5), BatchNorm(6, num_dims=4), nn.Sigmoid(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), BatchNorm(16, num_dims=4), nn.Sigmoid(),
    nn.MaxPool2d(kernel_size=2, stride=2), nn.Flatten(),
    nn.Linear(16*4*4, 120), BatchNorm(120, num_dims=2), nn.Sigmoid(),
    nn.Linear(120, 84), BatchNorm(84, num_dims=2), nn.Sigmoid(),
    nn.Linear(84, 10))
```

```
net[1].gamma.reshape((-1,)), net[1].beta.reshape((-1,))
```

```
(tensor([1.6752, 1.6357, 3.1687, 1.6708, 2.0469, 2.4485], device='cuda:0',
        grad_fn=<ViewBackward>),
 tensor([-0.2003, -0.8004,  2.2891,  0.5771,  0.3622, -0.3161], device='cuda:0',
        grad_fn=<ViewBackward>))
```

Implementation from Scratch

```python
class BatchNorm(nn.Module):
    # `num_features`: the number of outputs for a fully-connected layer
    # or the number of output channels for a convolutional layer. `num_dims`:
    # 2 for a fully-connected layer and 4 for a convolutional layer
    def __init__(self, num_features, num_dims):
        super().__init__()
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # The scale parameter and the shift parameter (model parameters) are
        # initialized to 1 and 0, respectively
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # The variables that are not model parameters are initialized to 0
        self.moving_mean = torch.zeros(shape)
        self.moving_var = torch.zeros(shape)

    def forward(self, X):
        # If `X` is not on the main memory, copy `moving_mean` and
        # `moving_var` to the device where `X` is located
        if self.moving_mean.device != X.device:
            self.moving_mean = self.moving_mean.to(X.device)
            self.moving_var = self.moving_var.to(X.device)
        # Save the updated `moving_mean` and `moving_var`
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma, self.beta, self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.9)
        return Y
```

Learning parameters

FC input
(batch × input)

CNN input
(batch × ch × p × q)

```python
def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Use `is_grad_enabled` to determine whether the current mode is training
    # mode or prediction mode
    if not torch.is_grad_enabled():
        # If it is prediction mode, directly use the mean and variance
        # obtained by moving average
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # When using a fully-connected layer, calculate the mean and
            # variance on the feature dimension
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # When using a two-dimensional convolutional layer, calculate the
            # mean and variance on the channel dimension (axis=1). Here we
            # need to maintain the shape of `X`, so that the broadcasting
            # operation can be carried out later
            mean = X.mean(dim=(0, 2, 3), keepdim=True)
            var = ((X - mean) ** 2).mean(dim=(0, 2, 3), keepdim=True)
        # In training mode, the current mean and variance are used for the
        # standardization
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # Update the mean and variance using moving average
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # Scale and shift
    return Y, moving_mean.data, moving_var.data
```

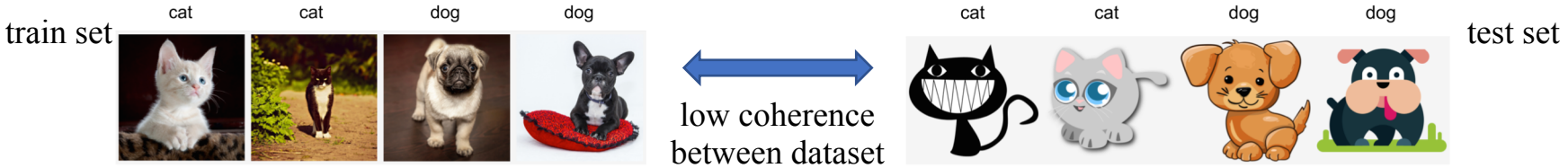
Prediction mode

Learning mode

Update the estimates by moving average

Controversy

- Covariate shift might be misnomer
  - Strictly speaking, covariate shift refers to **the change of distribution of input data** (not output)
  - Thus, the purpose of BN is called reducing '**internal** covariate shift'



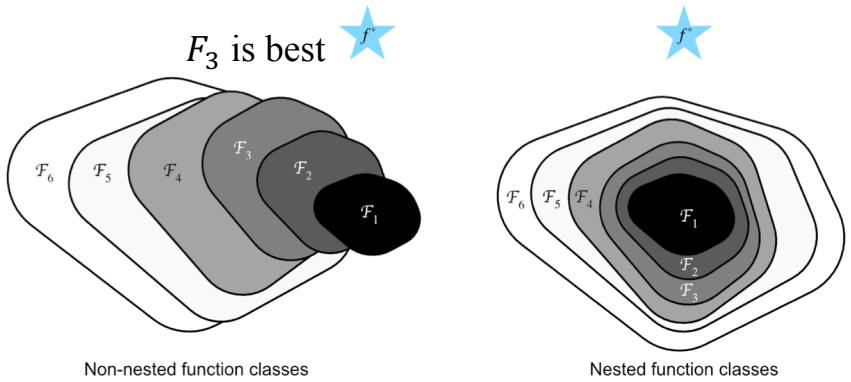
- No analytical explanations for the impact of batch normalization
- There are some clams that the major contribution of BN for learning is **not relieving internal covariate shift** but others, such as **preventing gradient explosion/vanishing** or **regularization**.
  - Ref: https://papers.nips.cc/paper/2018/file/905056c1ac1dad141560467e0a99e1cf-Paper.pdf

Intuition for Nested Architecture

- For deep earning function F, the objective of learning is finding optimal function with minimum loss.

$$f_{\mathcal{F}}^* \overset{\text{def}}{=} \underset{f}{\operatorname{argmin}}\, L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}.$$

- Even though more powerful or complex architecture F' is used, there is no guarantee that F' is better than F if $F \nsubseteq F'$.
- With nested relation between functions $F \subseteq F$, the issue can be avoided
  - If larger function contain the smaller ones, increasing complexity can guarantee increasing the expressive power of the network.



Residual Blocks

- $f(\mathbf{x}) = \mathbf{x} + \mathrm{g}(\mathrm{x})$
  - Dotted box: learning the residual mapping $f(\mathbf{x}) - \mathbf{x}$
  - Shortcut (residual) connection : directly feed the input $\mathbf{x}$ before activation
- For learning identity mapping $f(\mathbf{x}) = \mathbf{x}$, dotted box will learn to make its weights and biases become zero
- Requirements: output of botted box must have same dimension with input

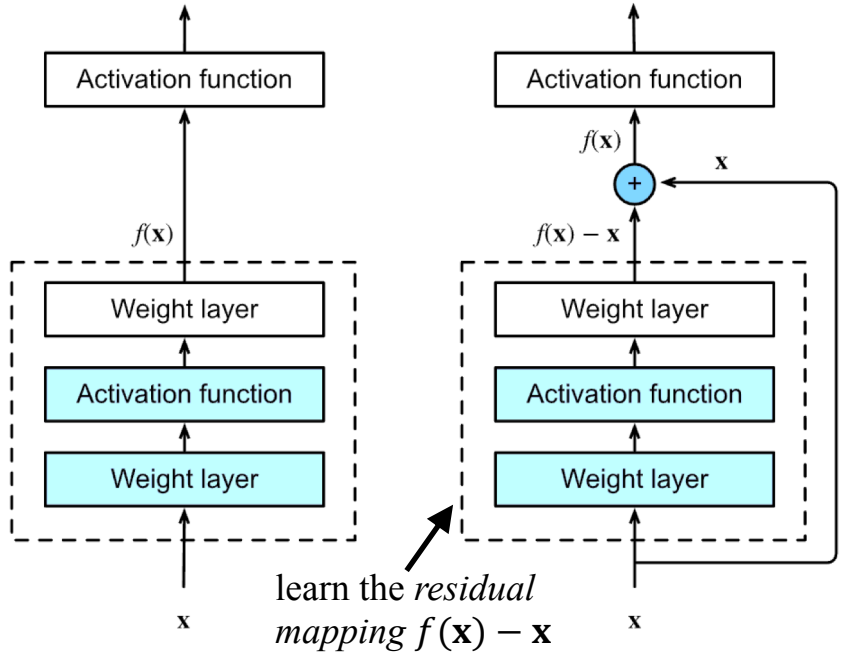


Fig. 7.6.2 A regular block (left) and a residual block (right).

Example

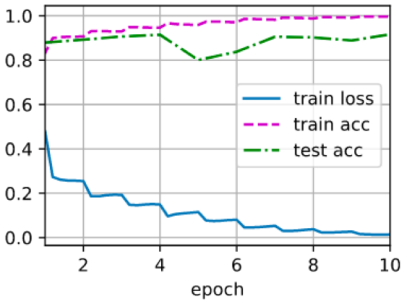
```
from d2l import torch as d2l
import torch
from torch import nn
from torch.nn import functional as F

class Residual(nn.Module):  #@save
    """The Residual block of ResNet."""
    def __init__(self, input_channels, num_channels,
                 use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.Conv2d(input_channels, num_channels,
                               kernel_size=3, padding=1, stride=strides)
        self.conv2 = nn.Conv2d(num_channels, num_channels,
                               kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(input_channels, num_channels,
                                   kernel_size=1, stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(num_channels)
        self.bn2 = nn.BatchNorm2d(num_channels)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```

Making input have same number of channel with intermediate layers

Shortcut (residual) connection



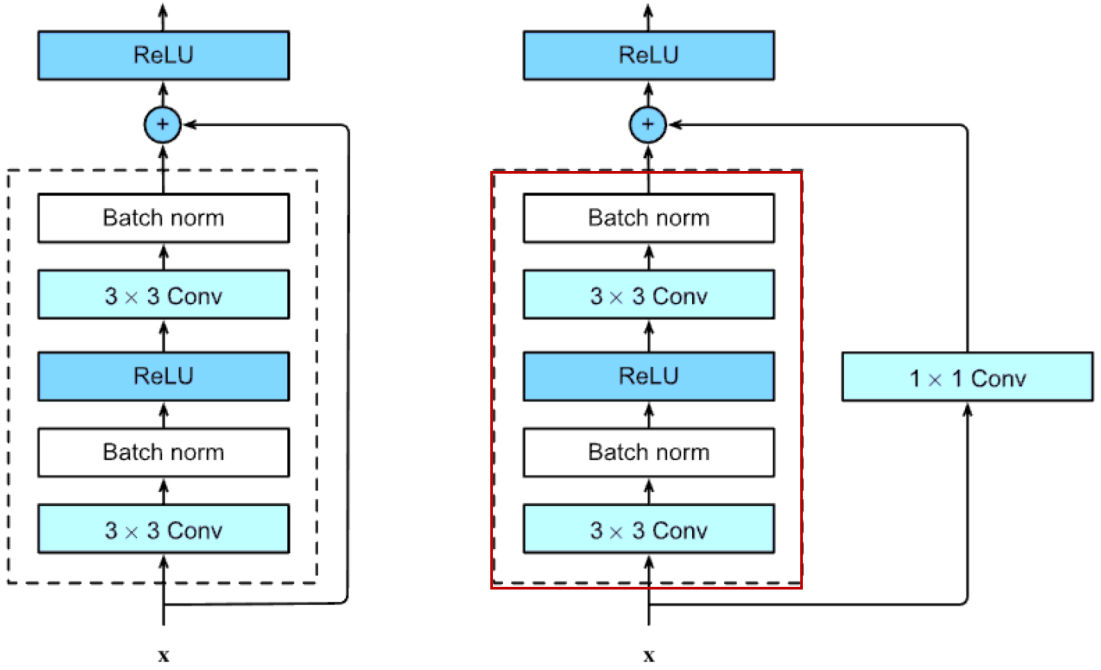
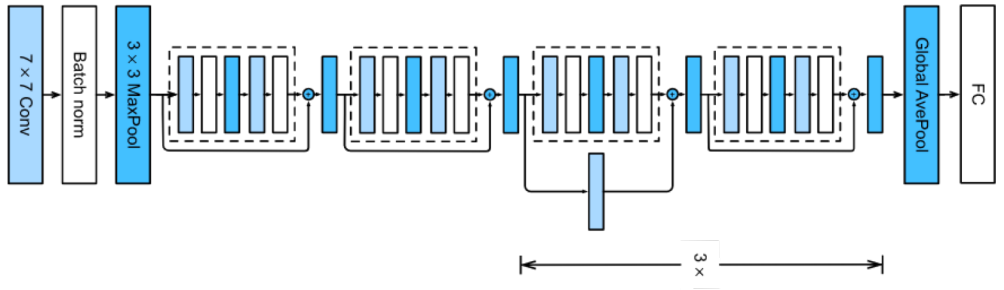


Fig. 7.6.3 ResNet block with and without 1×1 convolution.

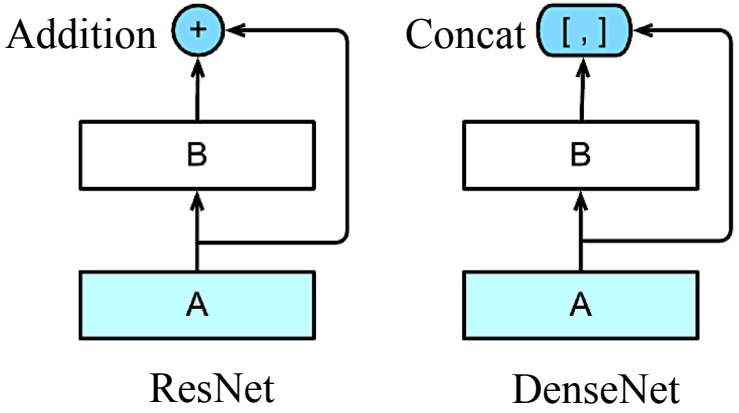


Full ResNet-18 Architecture

Mathematical interpretation of ResNet

wx + g(x)

- For Taylor expansion with x = 0, $f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 + \dots$
- For ResNet function, $f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x})$.
  - ResNet decomposes a function into a simple linear combinations with a more complex nonlinear ones



Dense Blocks

- **Concatenating** output sequence of each functions
  - The last layer of such a chain is densely connected to all previous layers.

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \dots] .$$

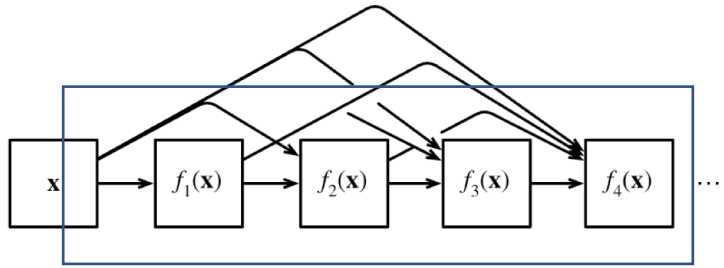


Fig. 7.7.2 Dense connections in DenseNet.

```python
def conv_block(input_channels, num_channels):
    return nn.Sequential(nn.BatchNorm2d(input_channels), nn.ReLU(),
        nn.Conv2d(input_channels, num_channels, kernel_size=3, padding=1))

class DenseBlock(nn.Module):
    def __init__(self, num_convs, input_channels, num_channels):
        super(DenseBlock, self).__init__()
        layer = []
        for i in range(num_convs):
            layer.append(conv_block(num_channels * i + input_channels, num_channels))
        self.net = nn.Sequential(*layer)

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            X = torch.cat((X, Y), dim=1)
            # Concatenate the input and output of each block on the channel dimension
        return X
```

```python
blk = DenseBlock(2, 3, 10)
X = torch.randn(4, 3, 8, 8)
Y = blk(X)
Y.shape
```

```
torch.Size([4, 23, 8, 8])
```

$X = [(4\ batch, 3\ channel, 8, 8)]$
$Y1 = [(4\ batch, 3\ channel, 8, 8), (4\ batch, 10\ channel, 8, 8)]$
$Y2 = [(4\ batch, 3\ channel, 8, 8), (4\ batch, 10\ channel, 8, 8), (4\ batch, 10\ channel, 8, 8)]$

Transition Layers

- Dense block can lead to an excessively complex model.
- Transition layer controls the complexity of the model
  - It reduces the number of channels by using the 1×1 convolutional layer and halves the height and width of the average pooling layer with a stride of 2

```
def transition_block(input_channels, num_channels):
    return nn.Sequential(
        nn.BatchNorm2d(input_channels), nn.ReLU(),
        nn.Conv2d(input_channels, num_channels, kernel_size=1),
        nn.AvgPool2d(kernel_size=2, stride=2))
```

```
blk = DenseBlock(2, 3, 10)
X = torch.randn(4, 3, 8, 8)
Y = blk(X)
Y.shape
```

→ `torch.Size([4, 23, 8, 8])` Dense block

```
blk = transition_block(23, 10)
blk(Y).shape
```

→ `torch.Size([4, 10, 4, 4])` Transition layer

```
Sequential(
  (0): Sequential(
    (0): Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  )
```

$1\times p \rightarrow 64\times(p/2)$

Transition Layer

```
  (2): Sequential(
    (0): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (1): ReLU()
    (2): Conv2d(192, 96, kernel_size=(1, 1), stride=(1, 1))
    (3): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
```

$192\times(p/2) \rightarrow 96\times(p/4)$

Dense Layer

```
  (1): DenseBlock(
    (net): Sequential(
      (0): Sequential(
        (0): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): ReLU()
        (2): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      )
      (1): Sequential(
        (0): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): ReLU()
        (2): Conv2d(96, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      )
      (2): Sequential(
        (0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): ReLU()
        (2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      )
      (3): Sequential(
        (0): BatchNorm2d(160, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): ReLU()
        (2): Conv2d(160, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      )
    )
  )
```

$64\times(p/2) \rightarrow [32\times(p/2), 64\times(p/2)] = 96\times(p/2)$

$96\times(p/2) \rightarrow [32\times(p/2), 96\times(p/2)] = 128\times(p/2)$

$128\times(p/2) \rightarrow [32\times(p/2), 128\times(p/2)] = 160\times(p/2)$

$160\times(p/2) \rightarrow [32\times(p/2), 160\times(p/2)] = 192\times(p/2)$

```
  (3): DenseBlock(
    (net): Sequential(
      (0): Sequential(
        (0): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): ReLU()
        (2): Conv2d(96, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      )
```

$96\times(p/2) \rightarrow [32\times(p/4), 96\times(p/4)] = 128\times(p/4)$