DSCI 565: MODERN CONVOLUTIONAL NEURAL NETWORKS

Ke-Thia Yao

Lecture 10: 2025-09-29

Batch Normalization

- For a minibatch $\mathcal{B}$ and instance $\mathrm{x} \in \mathcal{B}$, batch normalization

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

- Where γ, β are parameters (same shape as x) to be learned, and $\hat{\mu}_{\mathcal{B}}$ is minibatch mean, $\hat{\sigma}_{\mathcal{B}}$ minibatch standard deviation

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

- Where ϵ is a small constant to prevent division by zero

Why does Batch Normalization work?

- Scaling input data so that each feature has zero mean and unit variance is known to help control optimization
  - Maybe scaling data in internal layers would help as well
- During training, the hidden layer outputs can take values of widely varying magnitude, including across layers, across units in the same layer, across training time
  - This “drift” in distribution may hamper convergence
- Deep networks tend to overfit
  - Batch normalization acts like a regularizer by injecting noise

Batch Normalization

- Does not work well with very small batches
  - For flat layers with batch size of 1, the standard deviation is zero
  - Seems to work with sizes between 50 and 100
- The original paper places the batch normalization step between the affine transformation and the activation function, $\sigma\big(\mathrm{BN}(Wx + b)\big)$
  - Others have tried placing it after the activation function
- Once the model is trained
  - We calculate the mean and standard deviation using entire training set
  - Use these during prediction

Batch Normalization for Convolution

- A convolution scans across the image at multiple locations
- The mean and standard deviation is computed across all these locations
- If the convolution layer outputs $p * q$ elements, then average across all $p * q$ elements
- Batch Normalization is well defined even for batch size of 1

Layer Normalization

- Another type of normalization
- Normalizes each instance vector, such that the elements of the vector has zero mean and unit standard deviation
- With instance $\mathrm{x} = (x_1, \ldots, x_n)$, layer normalization LN is

$$\mathrm{x} \rightarrow \mathrm{LN}(\mathrm{x}) = \frac{\mathrm{x} - \hat{\mu}}{\hat{\sigma}}$$

- Where

$$\hat{\mu} \overset{\text{def}}{=} \frac{1}{n}\sum_{i=1}^{n} x_i \text{ and } \hat{\sigma}^2 \overset{\text{def}}{=} \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2 + \epsilon$$

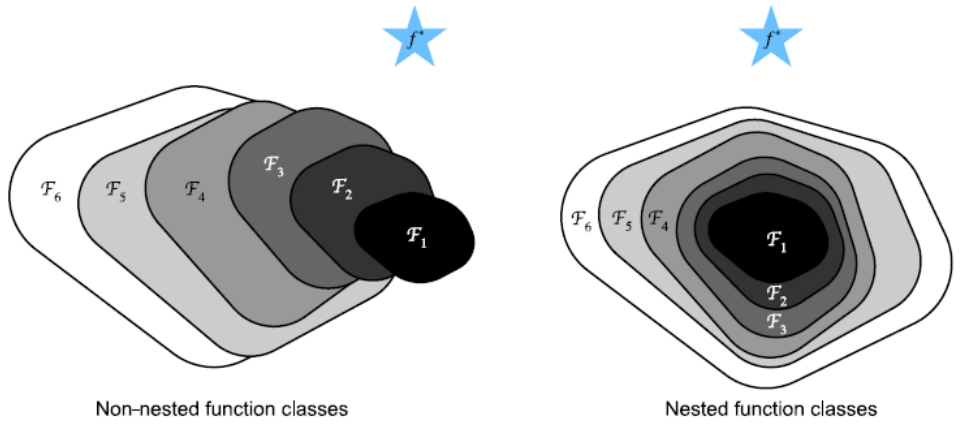
- Advantage:
  - Scale invariant $\mathrm{LN}(\mathrm{x}) \approx \mathrm{LN}(\alpha\mathrm{x})$
  - Does not depend on batch size

Notebook

- chapter_convolutional-modern/batch-norm.ipynb

Residual Networks (ResNet) and ResNeXt

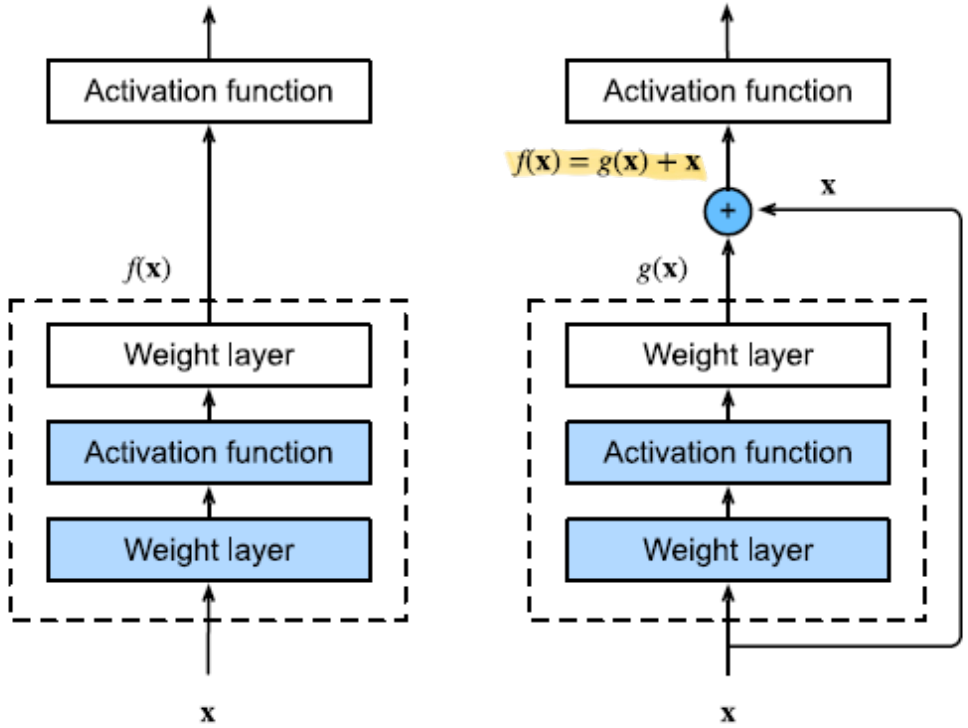
- Does adding additional layers necessarily increase the representational power of a deep network?
- Suppose the ground truth target function we want to find is f^*
- Further, suppose the set of possible function represented by our initial network is $\mathcal{F}_1$ and $f^* \notin \mathcal{F}_1$
- We add another layer to create $\mathcal{F}_2$. Is $\mathcal{F}_2$ any closer to f^*?
- If $\mathcal{F}_1 \not\subseteq \mathcal{F}_2$, then there is no guarantee
- How do we construct networks, such $\mathcal{F}_1 \subseteq \mathcal{F}_2$?



Residual Block

construct $F_1 \subseteq F_2 \subseteq F_3 \cdots$

- Each residual block outputs
  $f(\mathbf{x}) = g(\mathbf{x}) + \mathbf{x}$
- If $g(\mathbf{x}) = 0$, then the network outputs $\mathbf{x}$, the result of the previous block
- It is called residual block, because the block learns the residual
  $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$



- If can train newly added layer as an identity function, then new network is as effective as previous network

$F_1 \subseteq F_2$

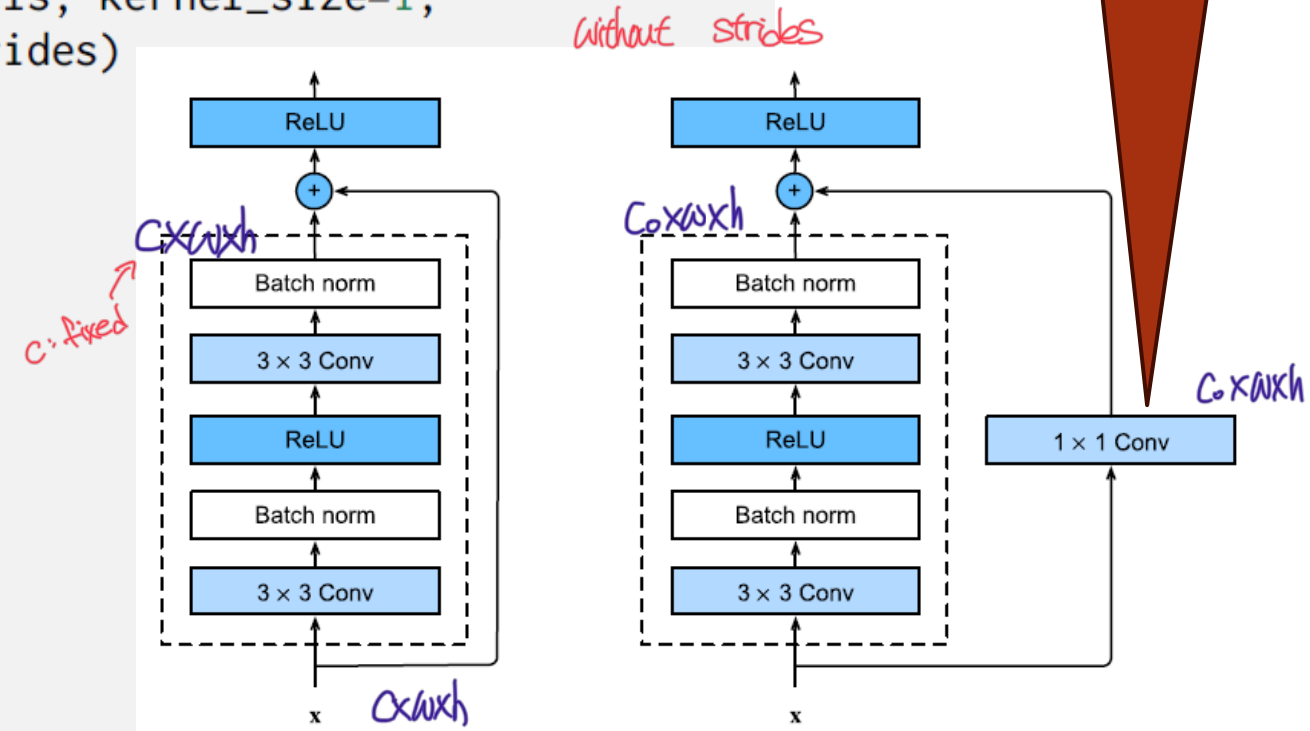
- Provides a quick way for input $\mathbf{x}$ to propagate up the network

Residual Block

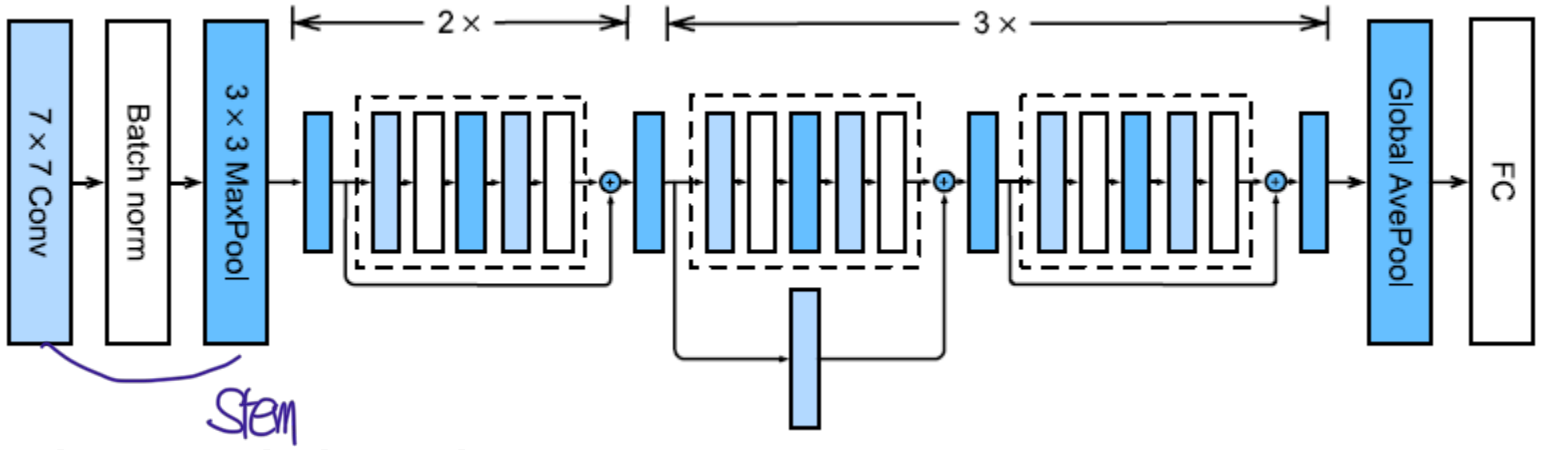
```python
class Residual(nn.Module):  #@save
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

    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)
```

Decrease the image size

Needed to make sure the input shape matches the output shape



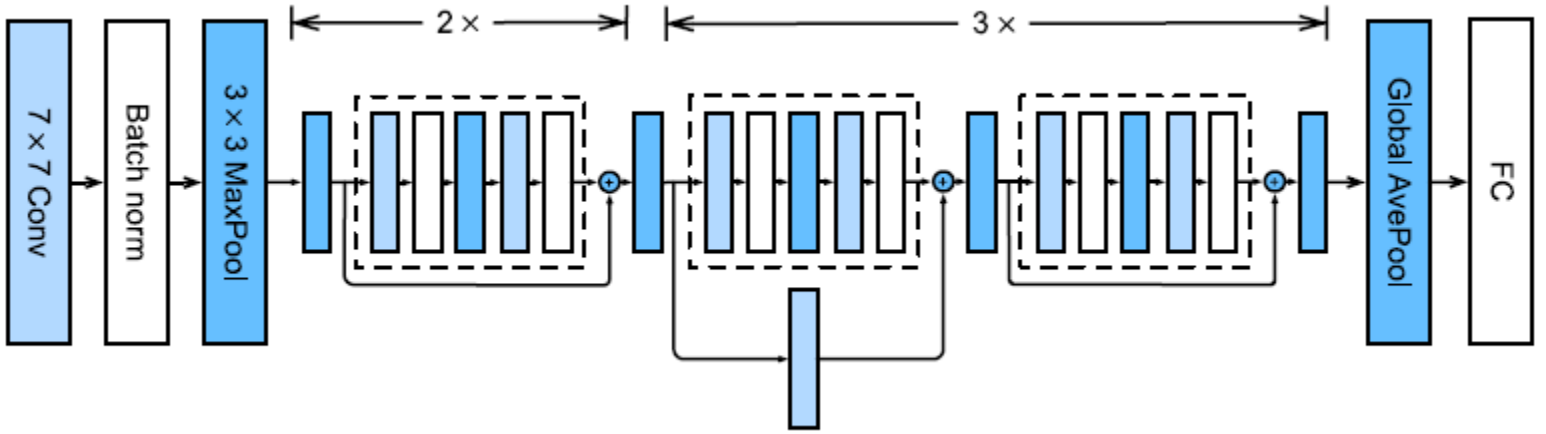
ResNet Model



- Stem, similar GoogLeNet but with batch normalization

```
class ResNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

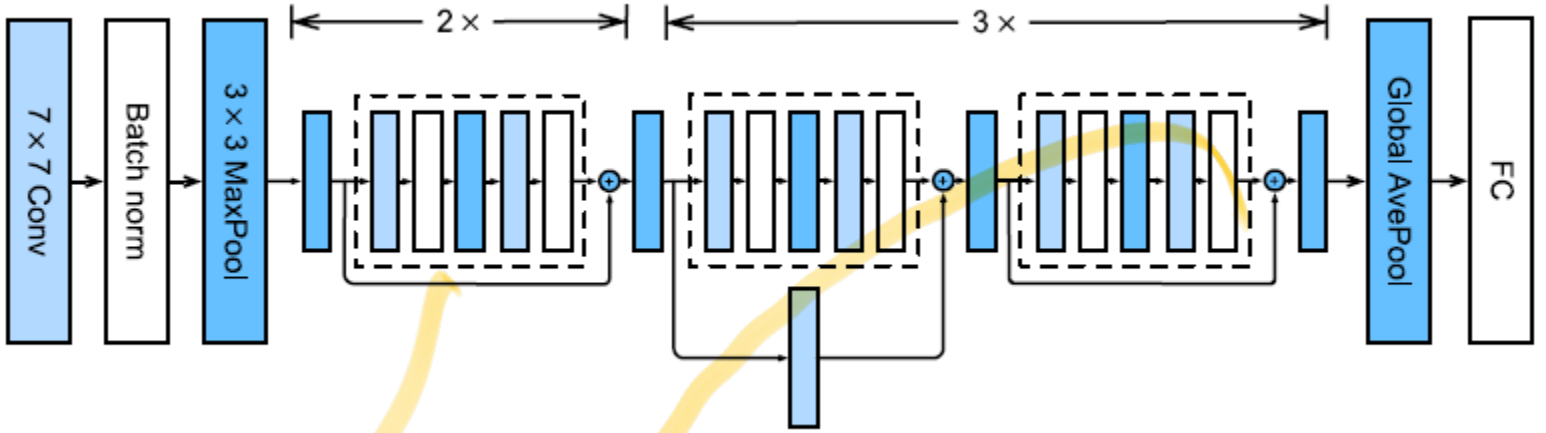
ResNet Model



- Method for adding multiple residual blocks

```
@d2l.add_to_class(ResNet)
def block(self, num_residuals, num_channels, first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels))
    return nn.Sequential(*blk)
```

ResNet18

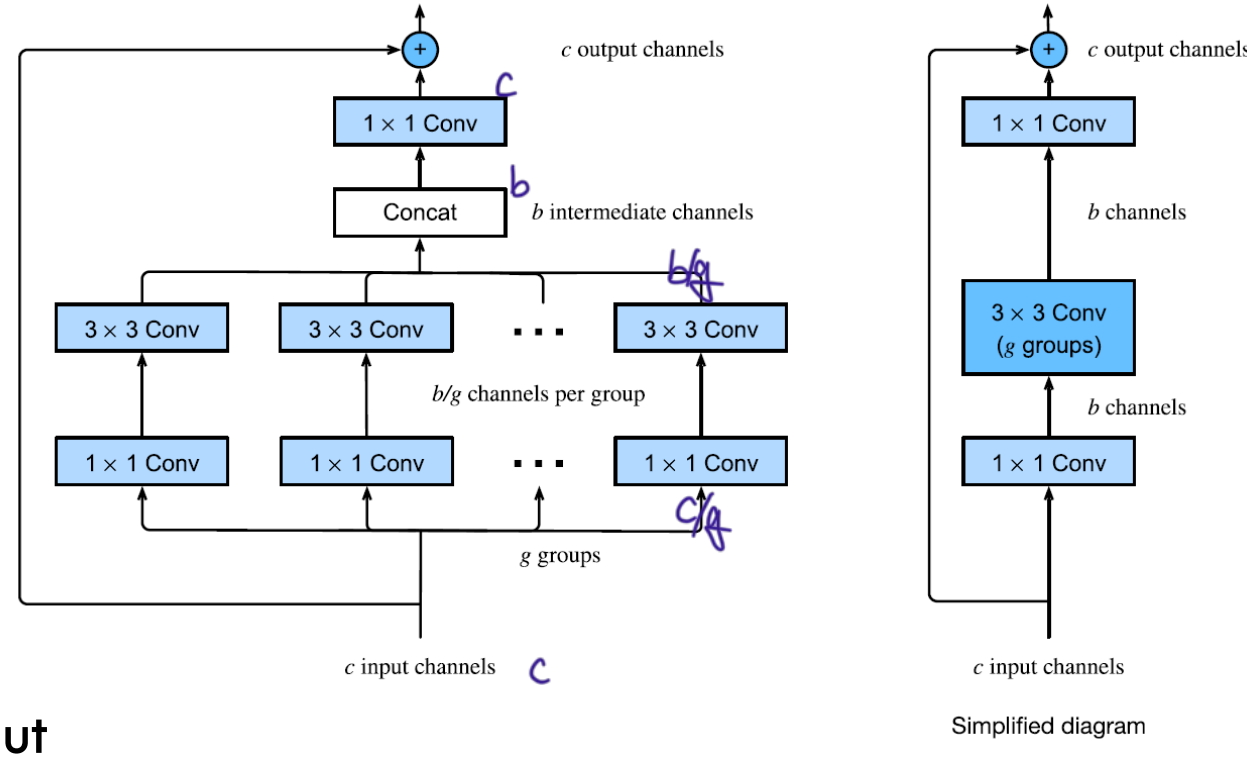



```
@d2l.add_to_class(ResNet)
def __init__(self, arch, lr=0.1, num_classes=10):
    super(ResNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.b1())
    for i, b in enumerate(arch):
        self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0)))
    self.net.add_module('last', nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
        nn.LazyLinear(num_classes)))
    self.net.apply(d2l.init_cnn)
```

```
class ResNet18(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                         lr, num_classes)
```

ResNeXt Block

- Like Inception net, each block contains multiple path, g groups
- Each group focus a subpart of the channels
- 1x1 Convs convert and divide channels into b/g channels per group
- Outputs of the g groups are concatenated to form bottleneck b channels
- The final 1x1 Conv converts into c_o output channels



Rationale for using groups

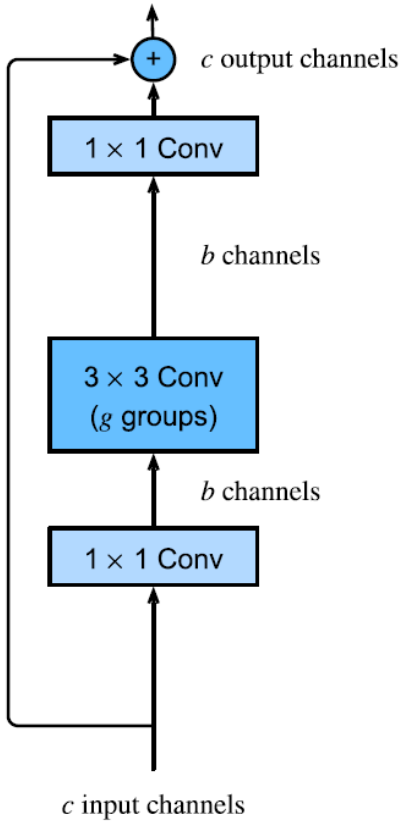
- Number of operation needed for convolutional layer of kernel size k is $\mathcal{O}(height * width * k^2 * c_i * c_o)$
- Focusing just on the input and output channels: $\mathcal{O}(c_i * c_o)$
- Using g groups each processing $\mathcal{O}\left(\frac{c_i}{g} * \frac{c_o}{g}\right)$ operations, we get

$$\mathcal{O}\left(g * \frac{c_i}{g} * \frac{c_o}{g}\right) = \mathcal{O}(c_i * c_o / g)$$

1/g

- Less computation
- For ResNeXt, if choose bottleneck $b < c_o$, then even less computation

ResNeXt Block



Simplified diagram

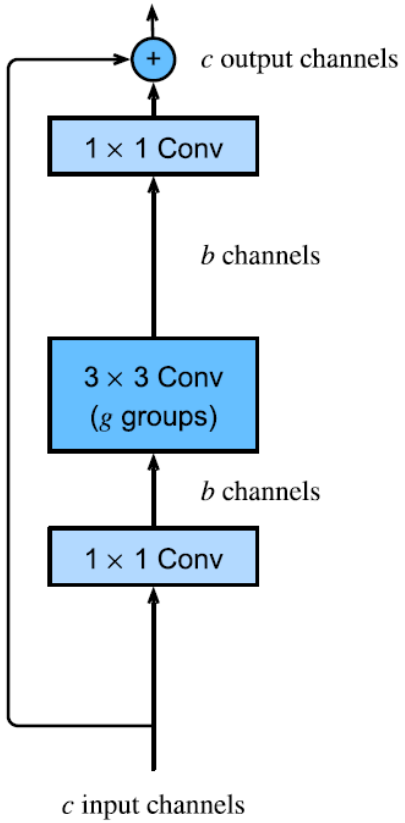
Output channels c_o

Groups g

Bottleneck channels:
$b = c_o * \text{bot\_mul}$

```
class ResNeXtBlock(nn.Module):  #@save
    """The ResNeXt block."""
    def __init__(self, num_channels, groups, bot_mul, use_1x1conv=False,
                 strides=1):
        super().__init__()
```

ResNeXt Block



Simplified diagram

```python
class ResNeXtBlock(nn.Module):  #@save
    """The ResNeXt block."""
    def __init__(self, num_channels, groups, bot_mul, use_1x1conv=False,
                 strides=1):
        super().__init__()
        bot_channels = int(round(num_channels * bot_mul))
        self.conv1 = nn.LazyConv2d(bot_channels, kernel_size=1, stride=1)
        self.conv2 = nn.LazyConv2d(bot_channels, kernel_size=3,
                                   stride=strides, padding=1,
                                   groups=bot_channels//groups)
        self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1, stride=1)
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()
        self.bn3 = nn.LazyBatchNorm2d()
        if use_1x1conv:
            self.conv4 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
            self.bn4 = nn.LazyBatchNorm2d()
        else:
            self.conv4 = None

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = F.relu(self.bn2(self.conv2(Y)))
        Y = self.bn3(self.conv3(Y))
        if self.conv4:
            X = self.bn4(self.conv4(X))
        return F.relu(Y + X)
```

???

Densely Connected Networks (DenseNet)

- ResNet can be thought of as decomposing a function into an identity function and a higher order function:

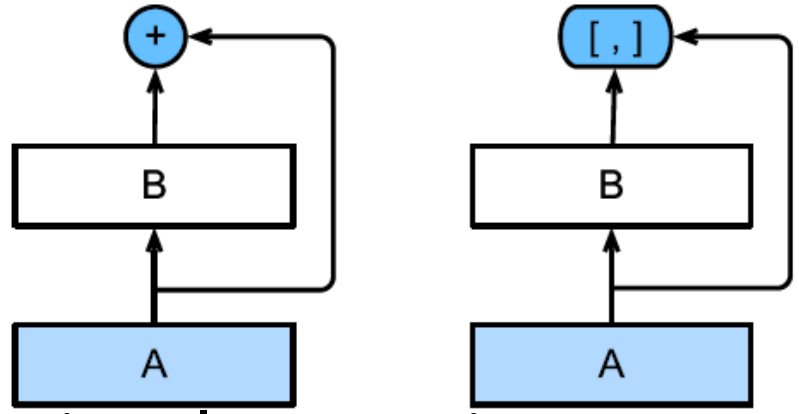
$$f(x) = x + g(x)$$

- Can we go beyond two terms?
- Using Taylor expansion as an analogy:

$$f(x) = f(0) + x \cdot \left[f'(0) + x \cdot \left[\frac{f''(0)}{2!} + x \cdot \left[\frac{f'''(0)}{3!} + \cdots \right] \right] \right]$$

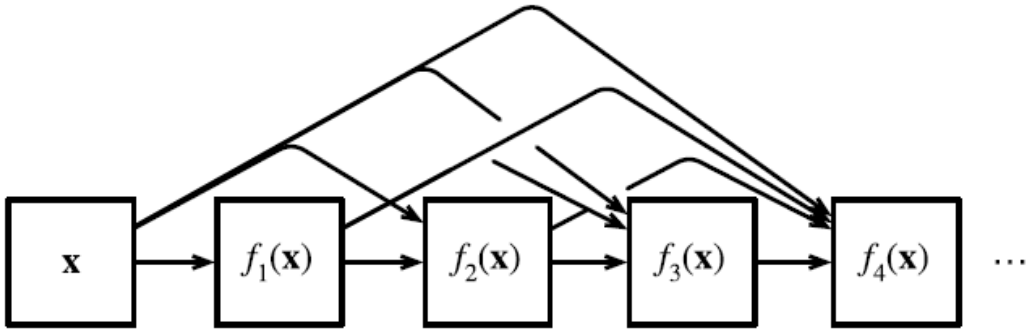
DenseNet Blocks

- Instead of summing, let's concatenate channels

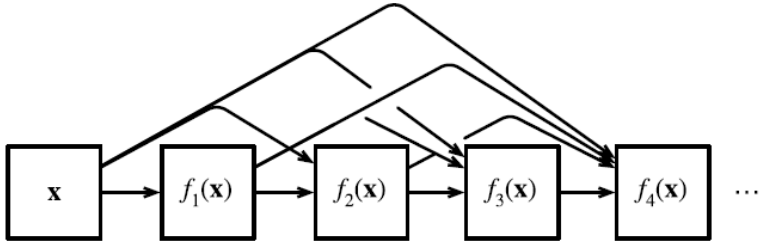


- Each layer takes the concatenation from the previous layer as input

$$\mathbf{x} \rightarrow [\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \ldots]$$



DenseNet Blocks




```
def conv_block(num_channels):
    return nn.Sequential(
        nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=3, padding=1))
```

```
class DenseBlock(nn.Module):
    def __init__(self, num_convs, num_channels):
        super(DenseBlock, self).__init__()
        layer = []
        for i in range(num_convs):
            layer.append(conv_block(num_channels))
        self.net = nn.Sequential(*layer)
    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            # Concatenate input and output of each block along the channels
            X = torch.cat((X, Y), dim=1)
        return X
```

Override forward()

For each iteration, the number of channels increase by num_channels

Transition Layers

DenseBlock with 2 Conv Layers

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

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

Transition Layer

- Reduce channels and image shape

```
def transition_block(num_channels):
    return nn.Sequential(
        nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=1),
        nn.AvgPool2d(kernel_size=2, stride=2))
```

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

```
torch.Size([4, 10, 4, 4])
```

DenseNet Model


```
class DenseNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

DenseNet Model

```
@d2l.add_to_class(DenseNet)
def __init__(self, num_channels=64, growth_rate=32, arch=(4, 4, 4, 4),
             lr=0.1, num_classes=10):
    super(DenseNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.b1())
    for i, num_convs in enumerate(arch):
        self.net.add_module(f'dense_blk{i+1}', DenseBlock(num_convs,
                                                          growth_rate))
        # The number of output channels in the previous dense block
        num_channels += num_convs * growth_rate
        # A transition layer that halves the number of channels is added
        # between the dense blocks
        if i != len(arch) - 1:
            num_channels //= 2
            self.net.add_module(f'tran_blk{i+1}', transition_block(
                num_channels))
    self.net.add_module('last', nn.Sequential(
        nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
        nn.LazyLinear(num_classes)))
    self.net.apply(d2l.init_cnn)
```