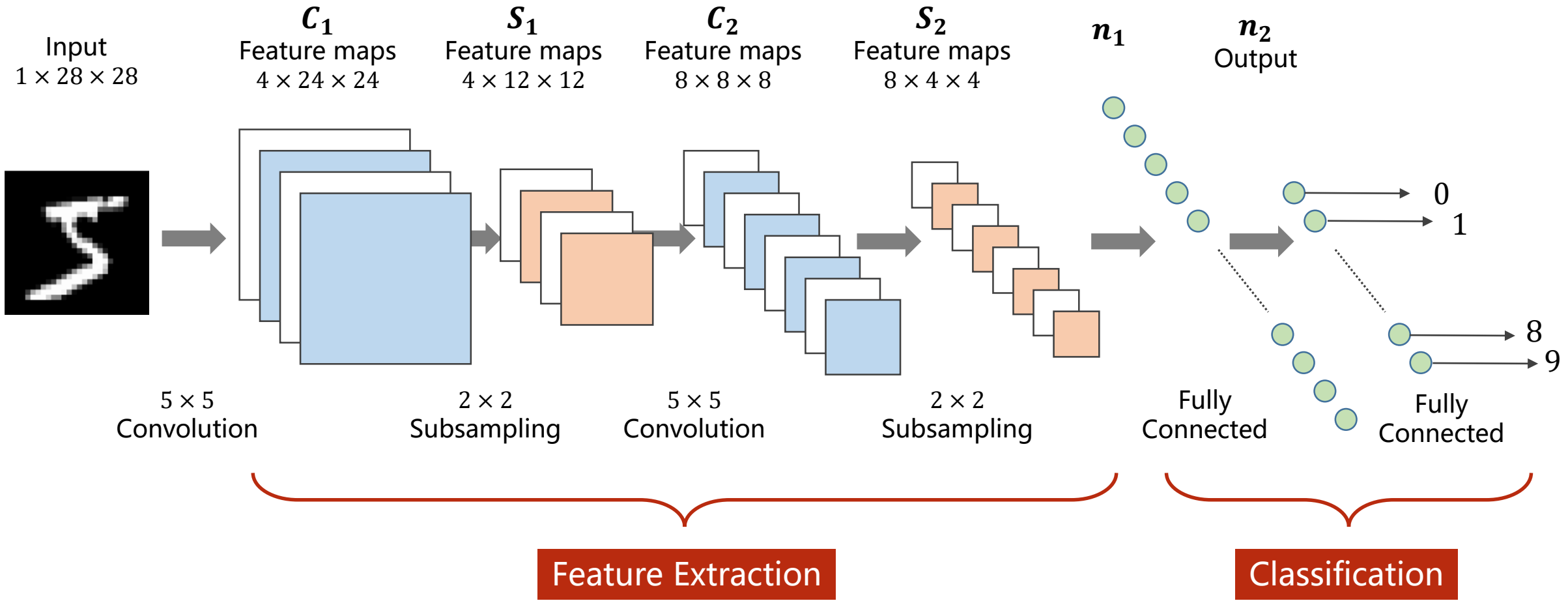
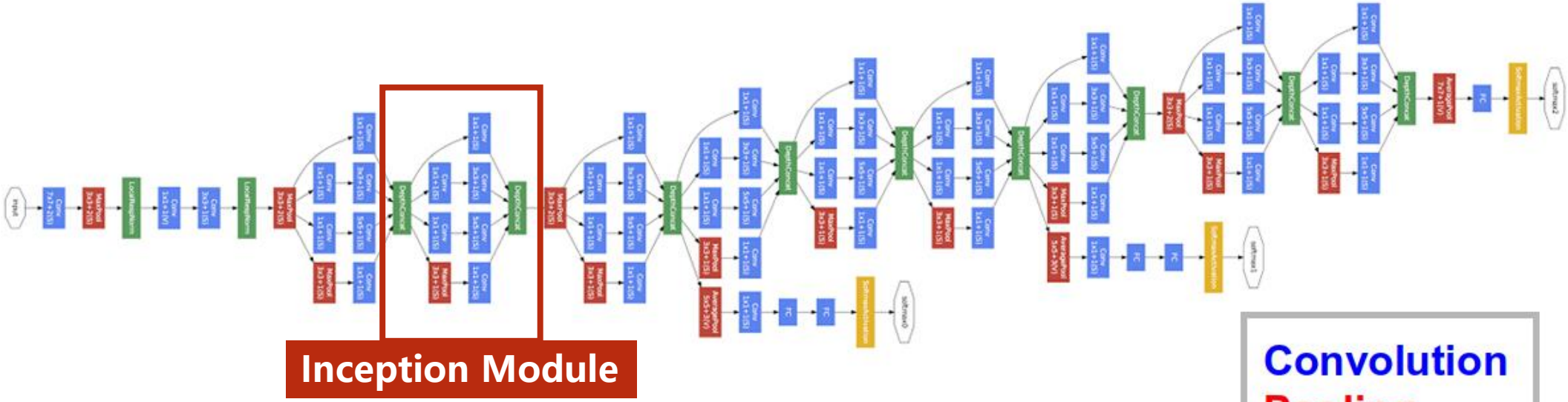
PyTorch Tutorial

11. Advanced CNN

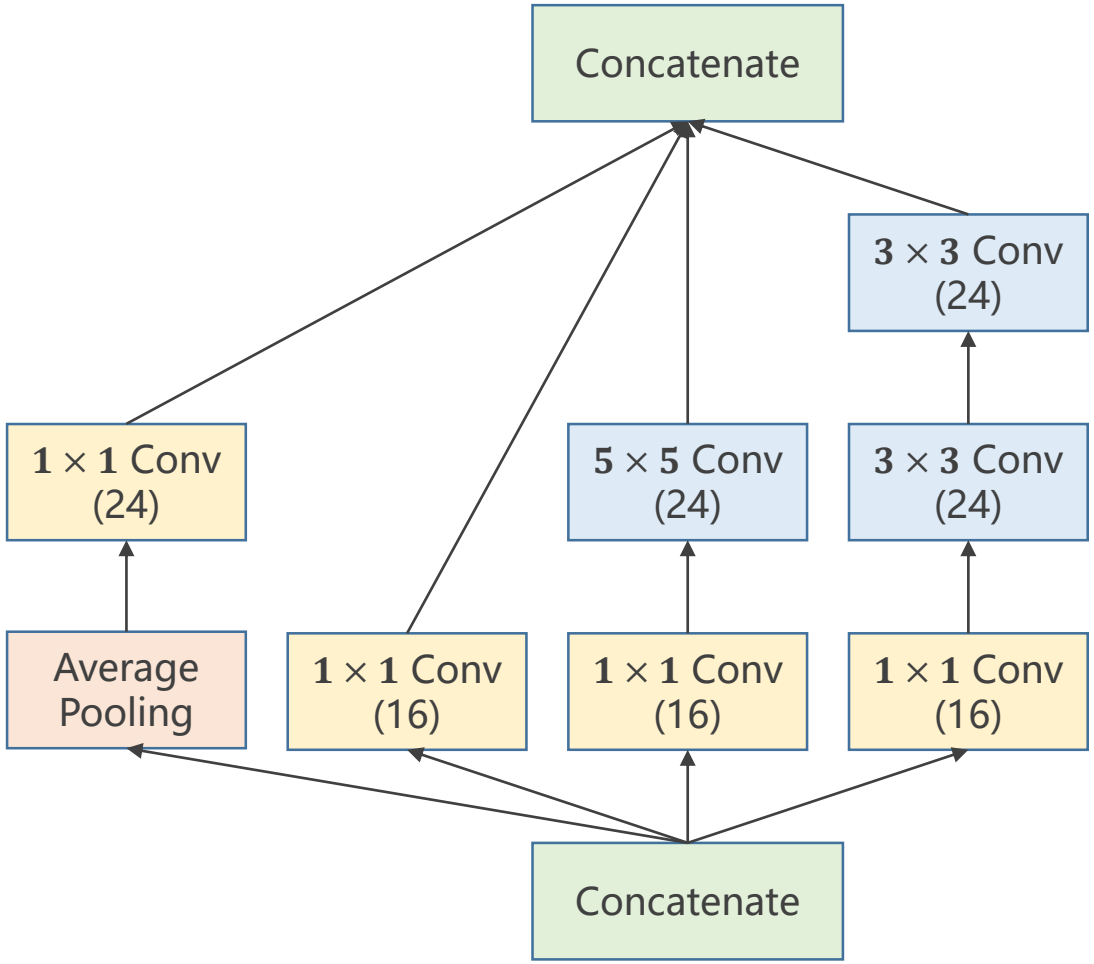
Revision

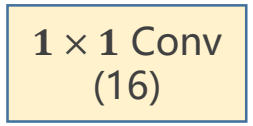

Input
1 × 28 × 28
C_1
Feature maps
4 × 24 × 24
S_1
Feature maps
4 × 12 × 12
C_2
Feature maps
8 × 8 × 8
S_2
Feature maps
8 × 4 × 4
n_1
n_2
Output
0
1
8
9
5 × 5
Convolution
2 × 2
Subsampling
5 × 5
Convolution
2 × 2
Subsampling
Fully
Connected
Fully
Connected
Feature Extraction
Classification

GoogLeNet



Inception Module

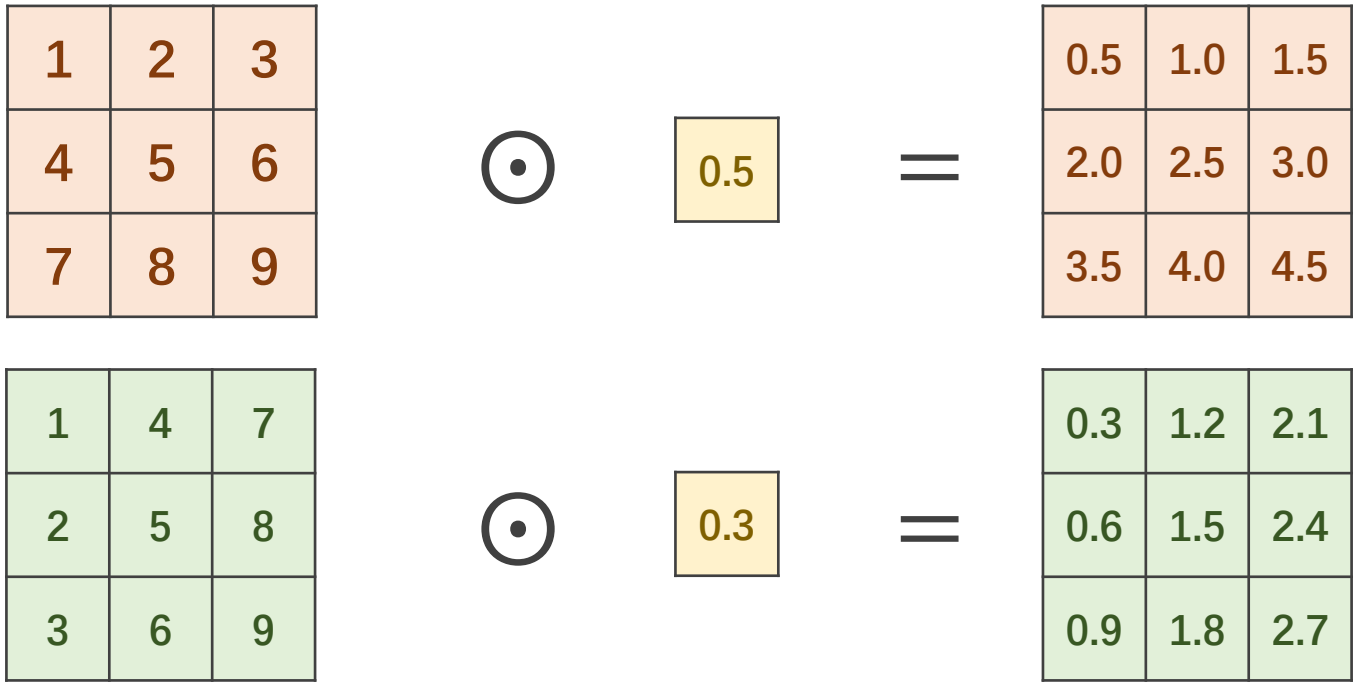




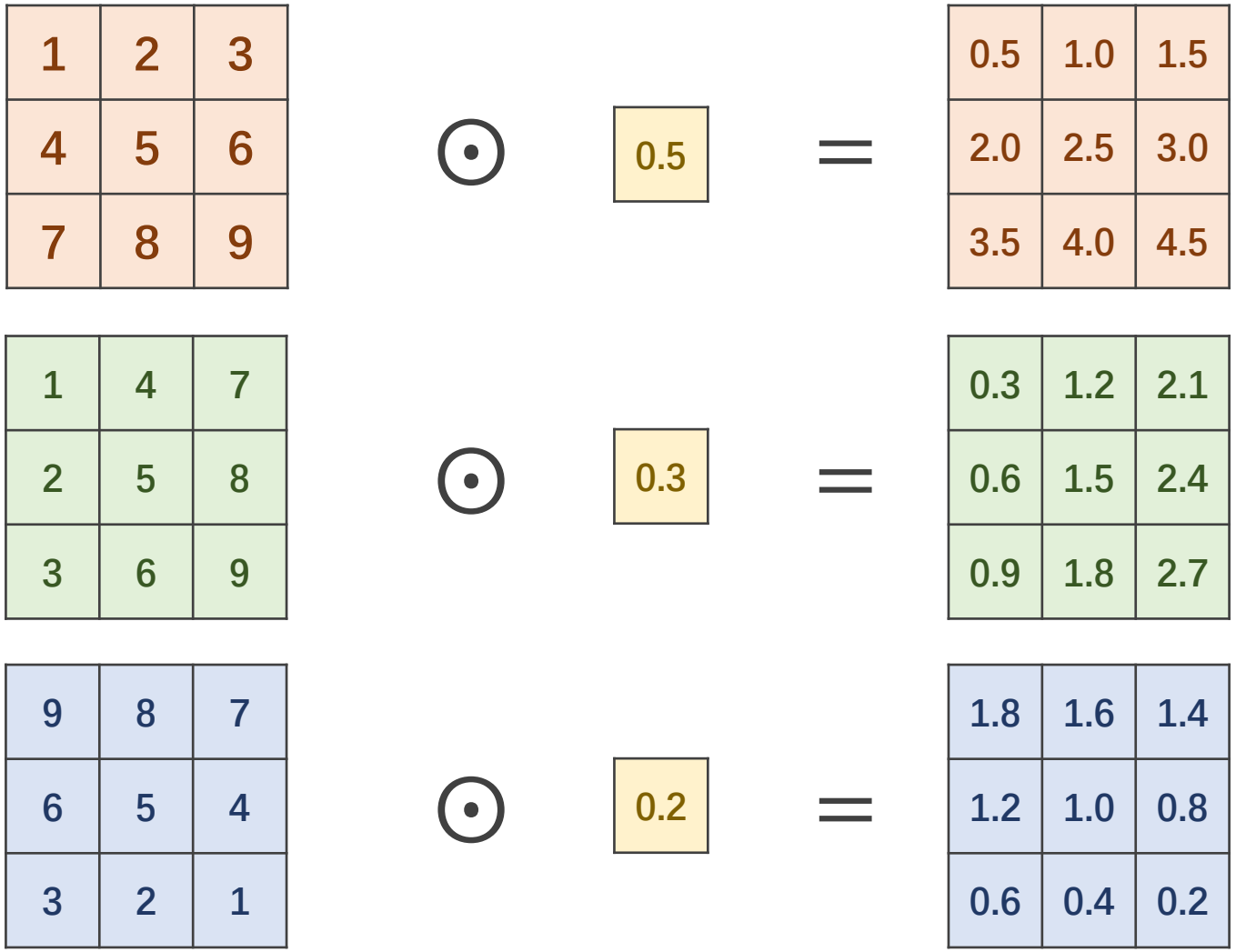
What is 1×1 convolution?

What is 1x1 convolution?

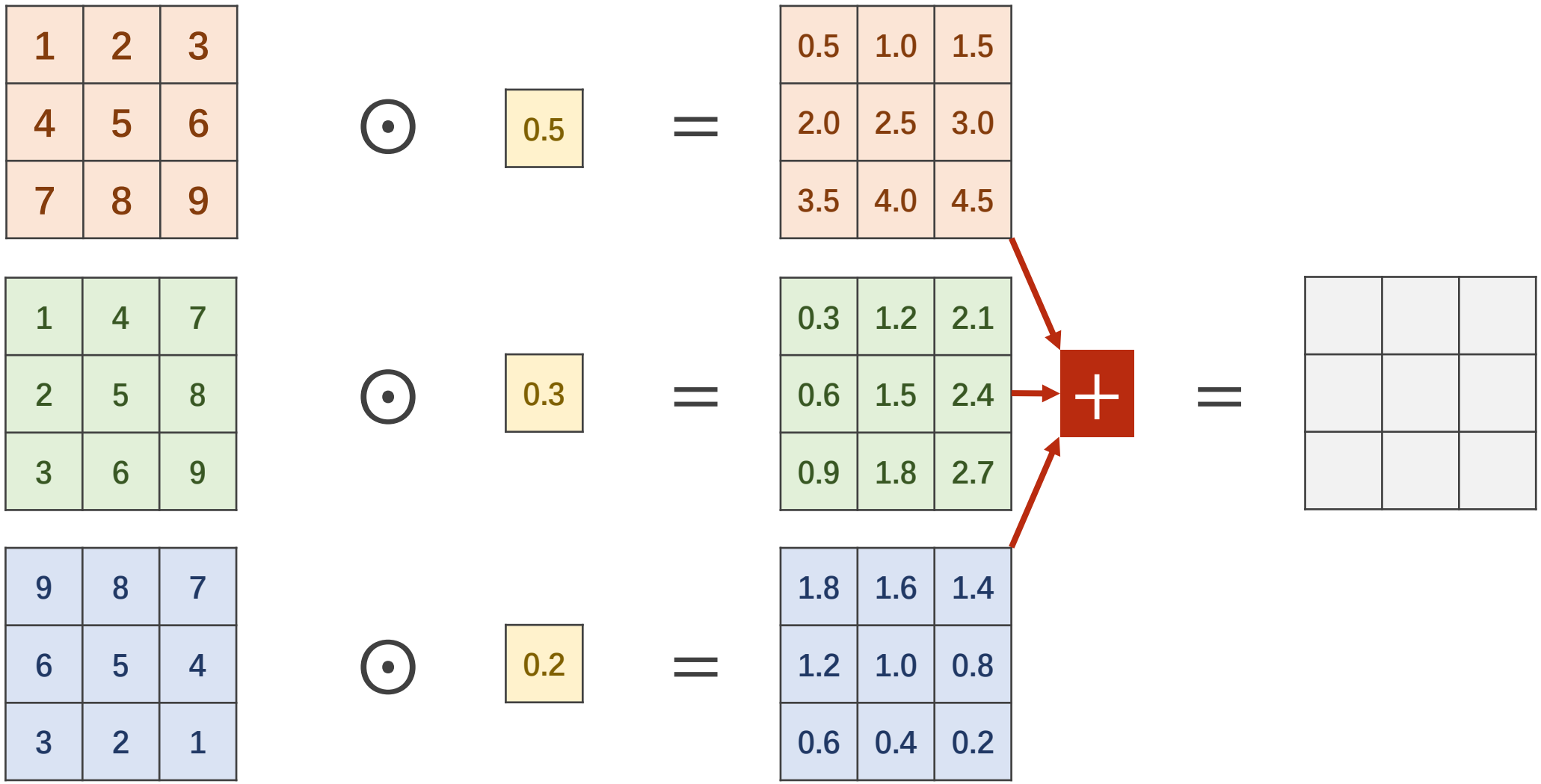
What is 1x1 convolution?



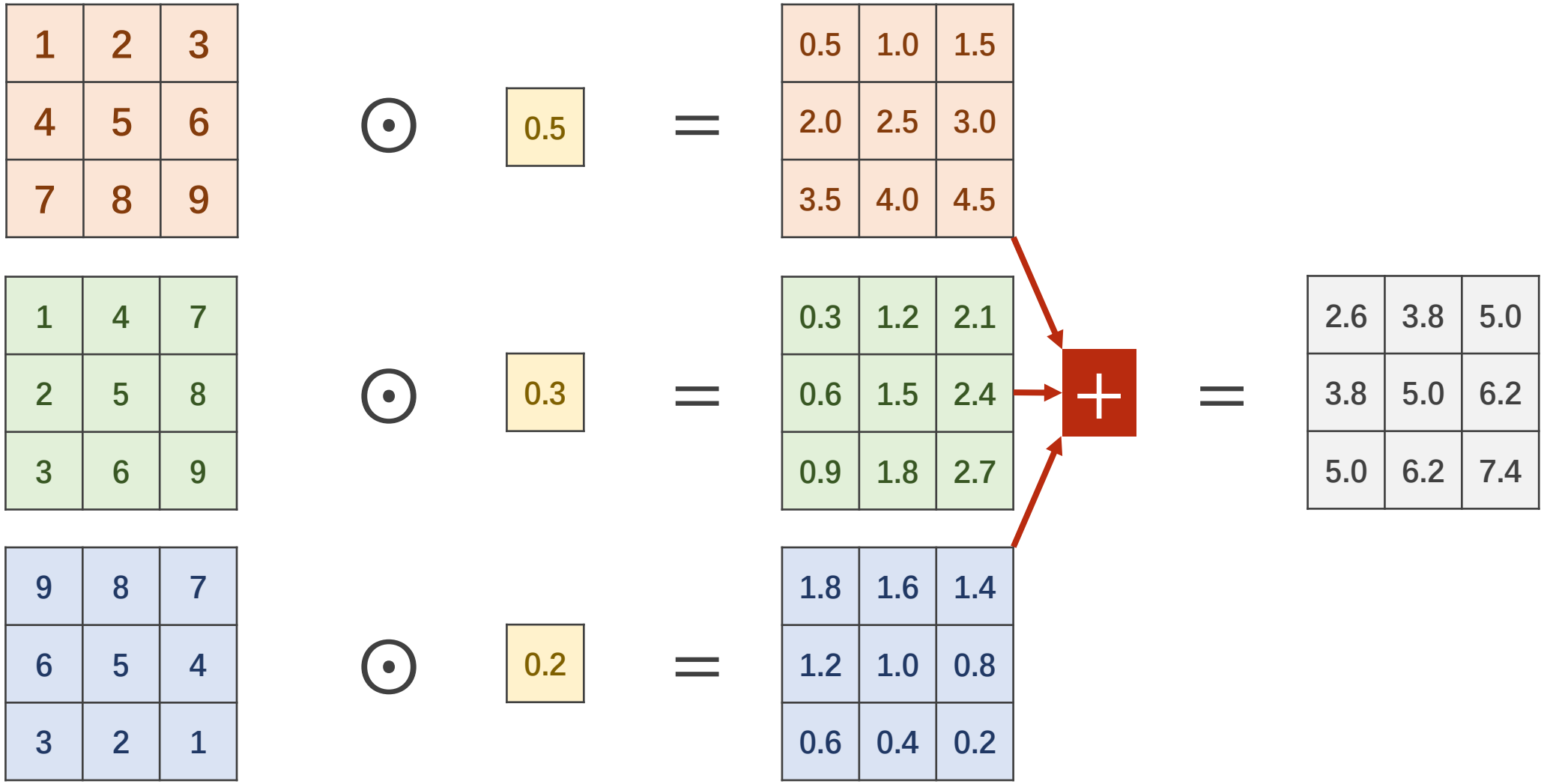
What is 1x1 convolution?



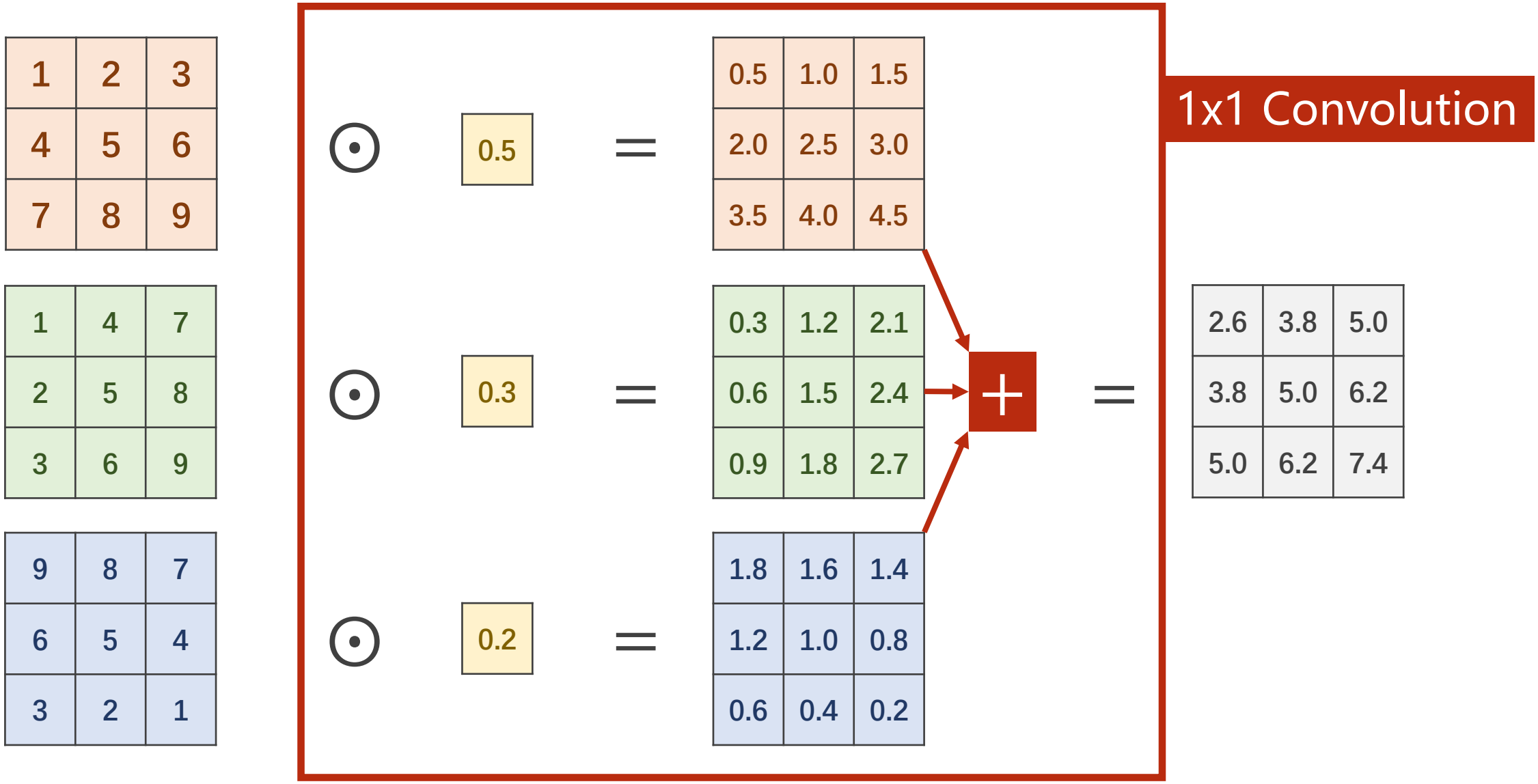
What is 1x1 convolution?



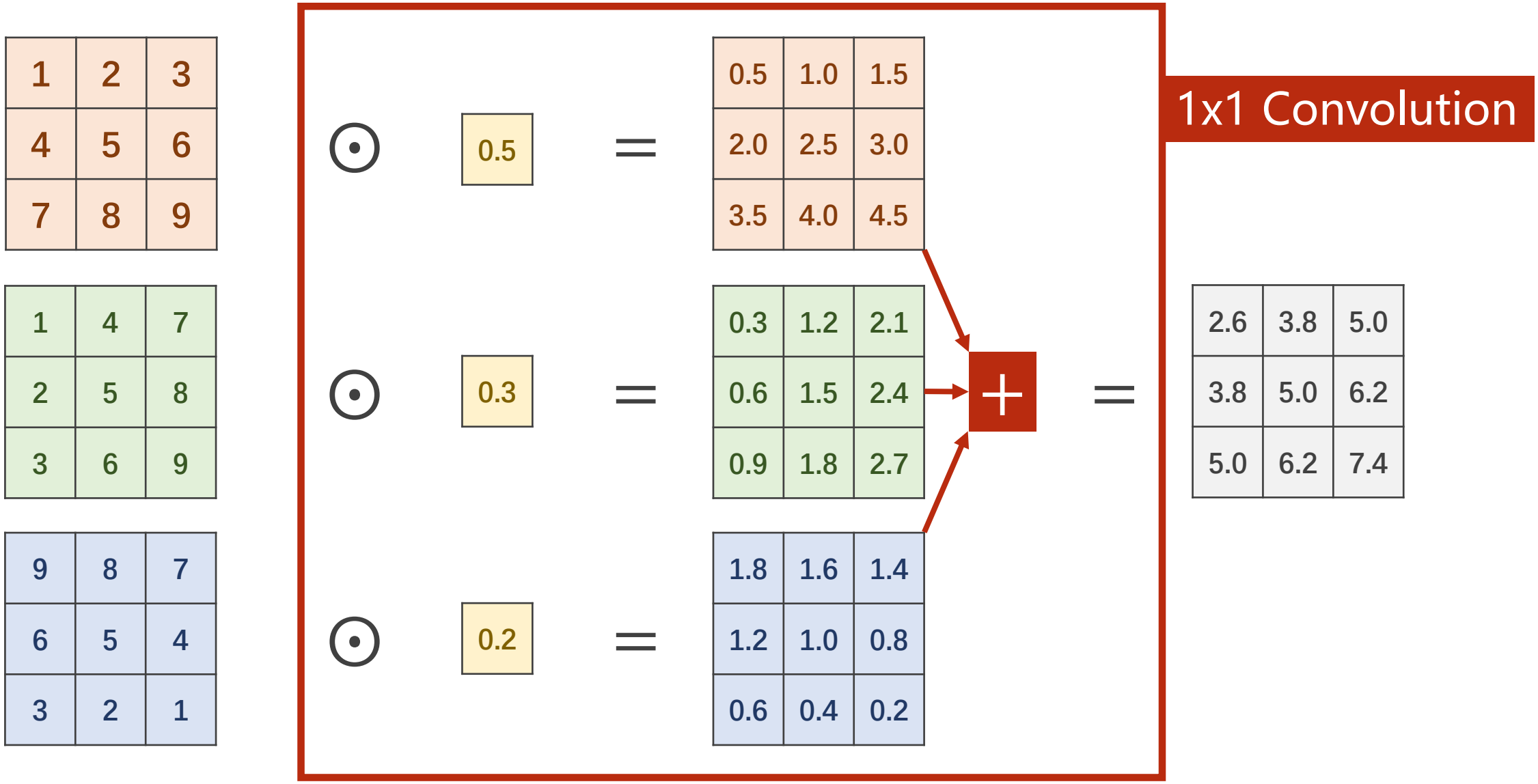
What is 1x1 convolution?



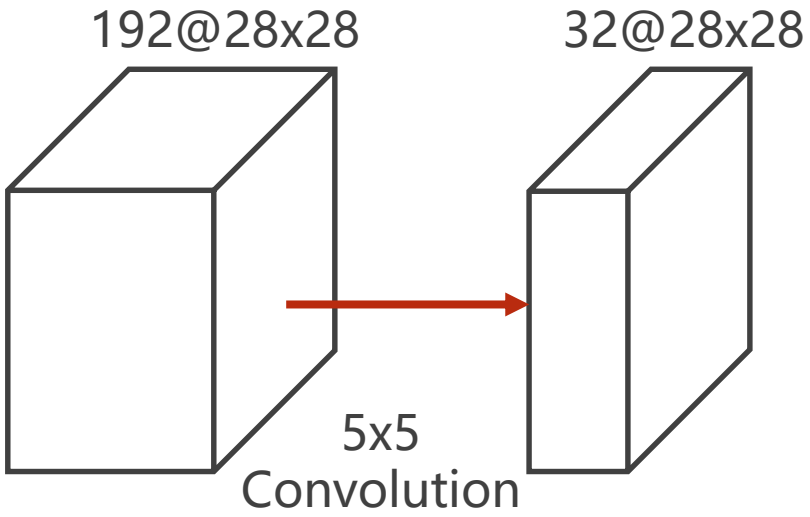
What is 1x1 convolution?



What is 1x1 convolution?



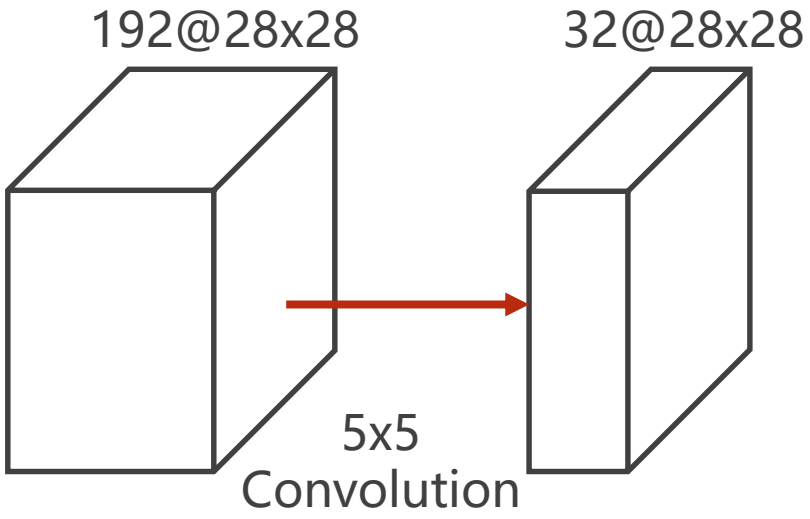
Why is 1x1 convolution?



Operations:

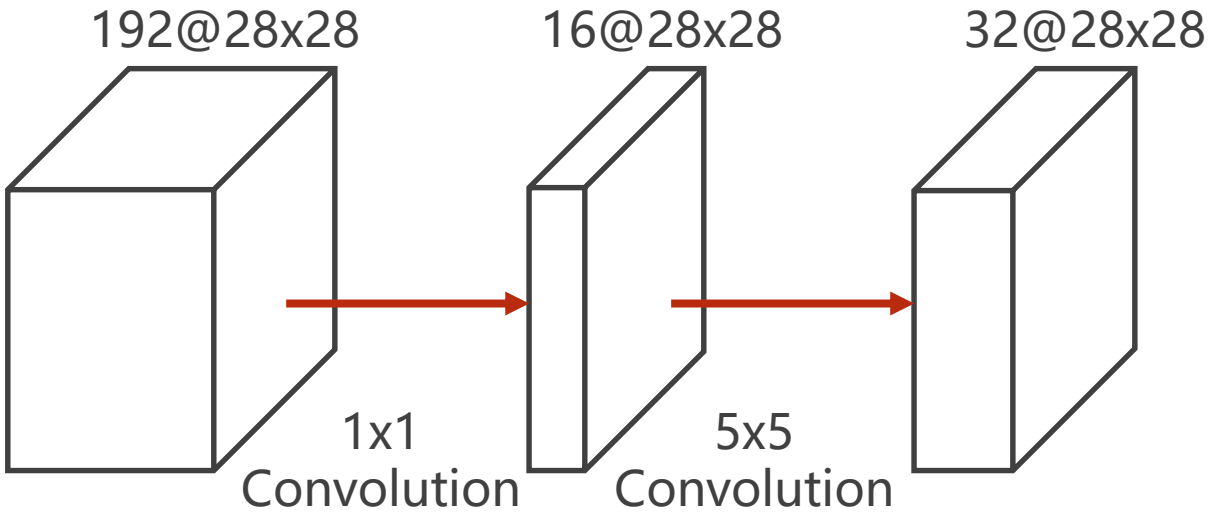
$5^2 \times 28^2 \times 192 \times 32 = 120{,}422{,}400$

Why is 1x1 convolution?



Operations:

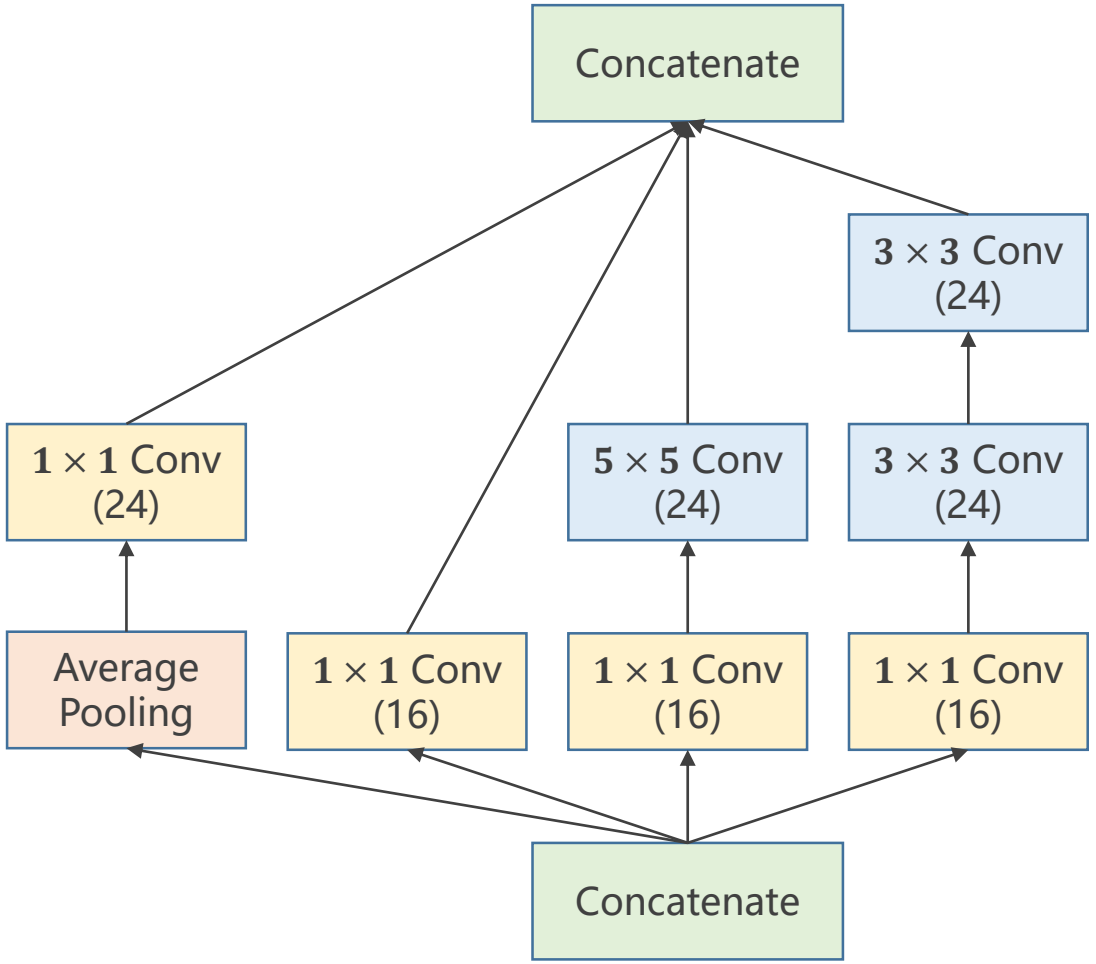
$5^2 \times 28^2 \times 192 \times 32 =$
$120{,}422{,}400$



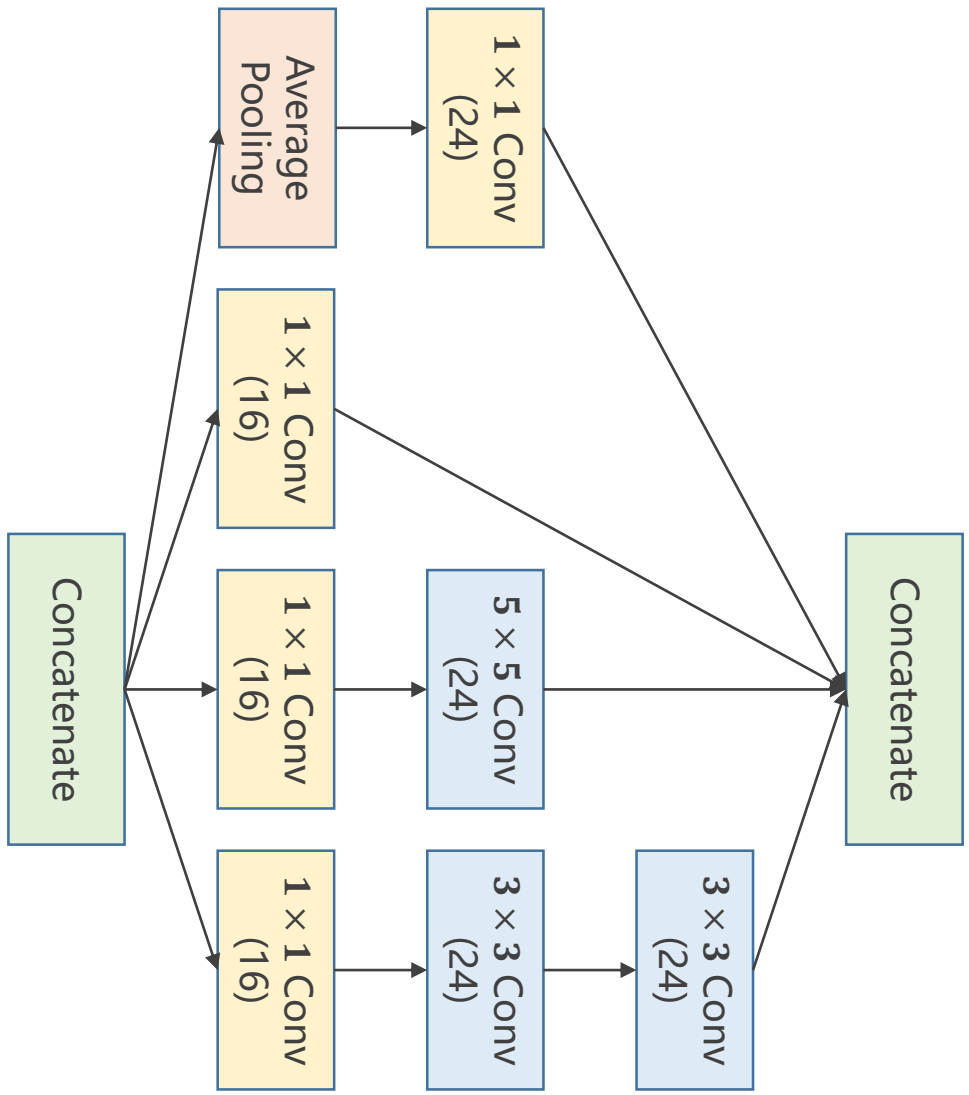
Operations:

$1^2 \times 28^2 \times 192 \times 16 +$
$5^2 \times 28^2 \times 16 \times 32 =$
$12{,}433{,}648$

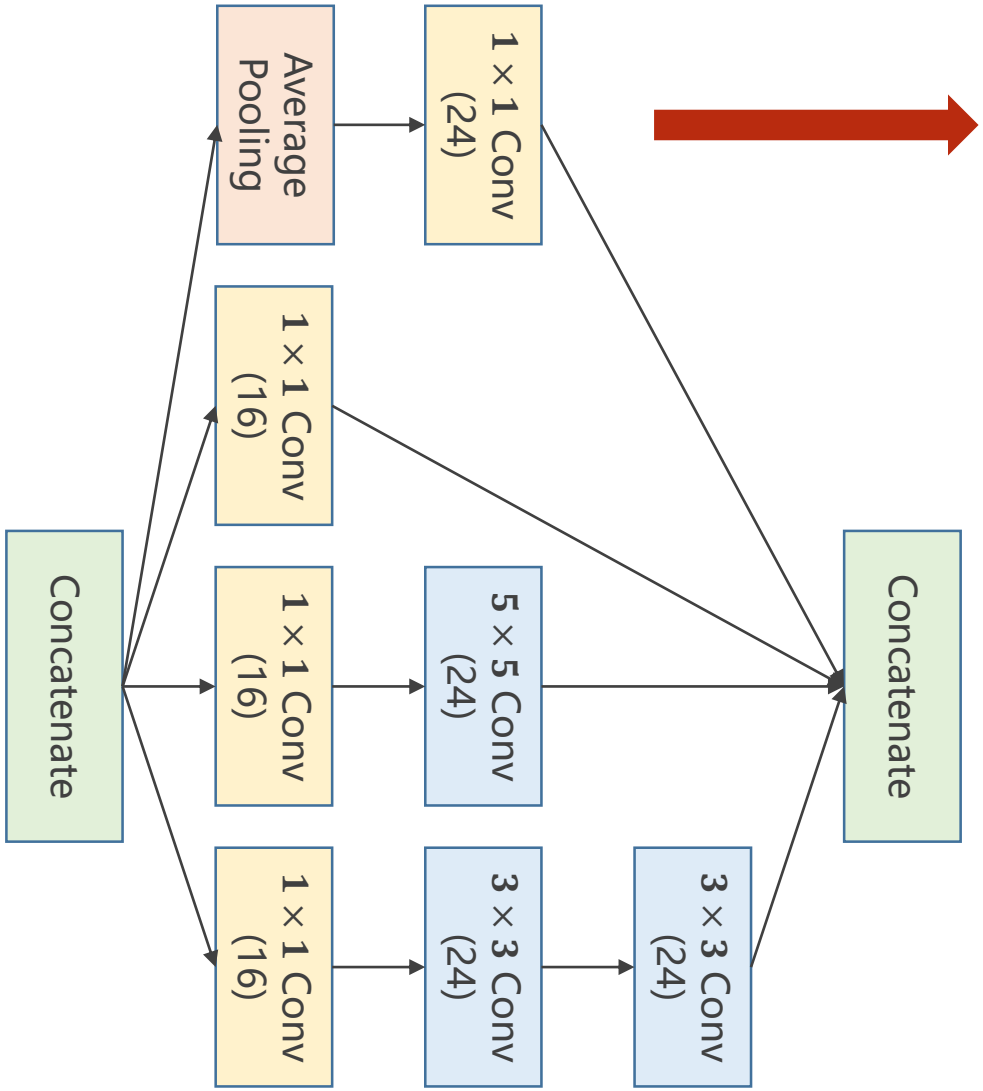
Implementation of Inception Module



Implementation of Inception Module

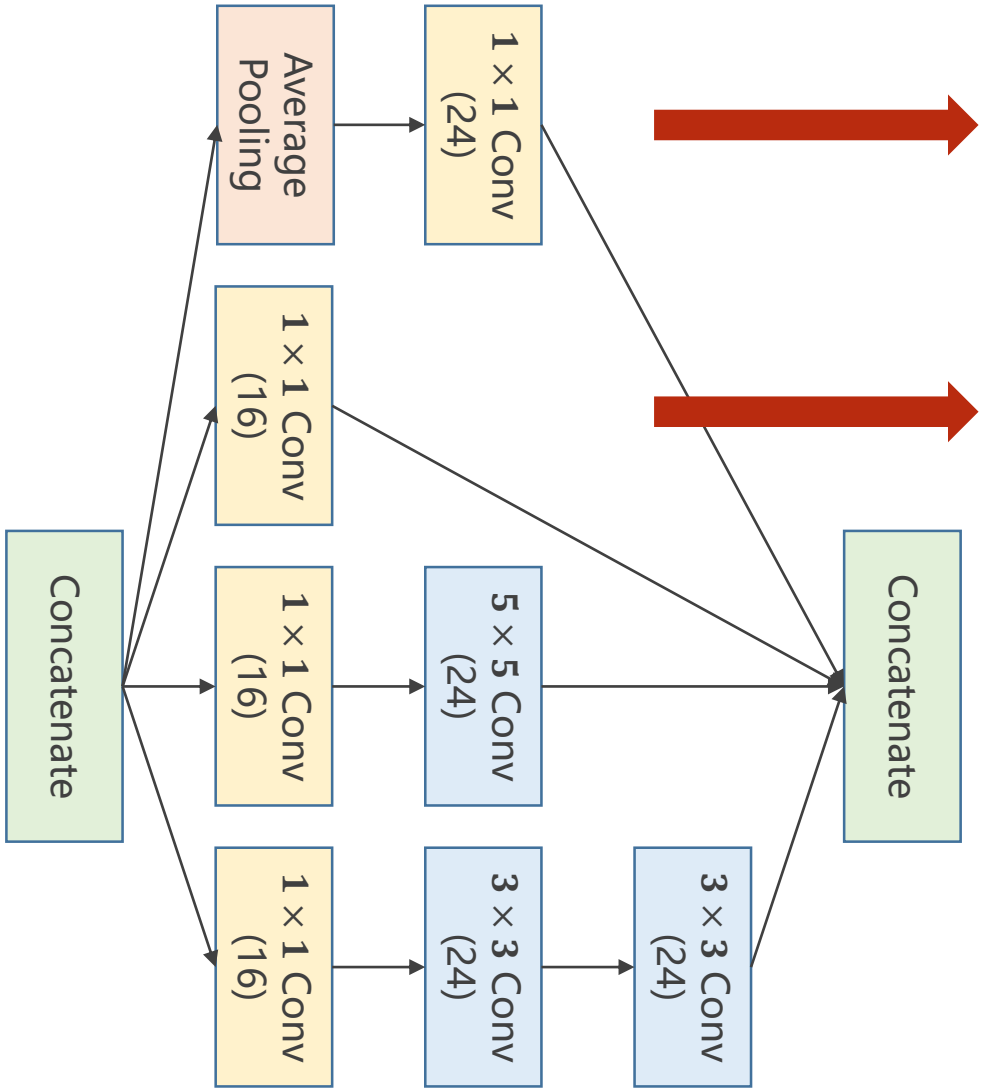


Implementation of Inception Module




```
self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
branch_pool = self.branch_pool(branch_pool)
```

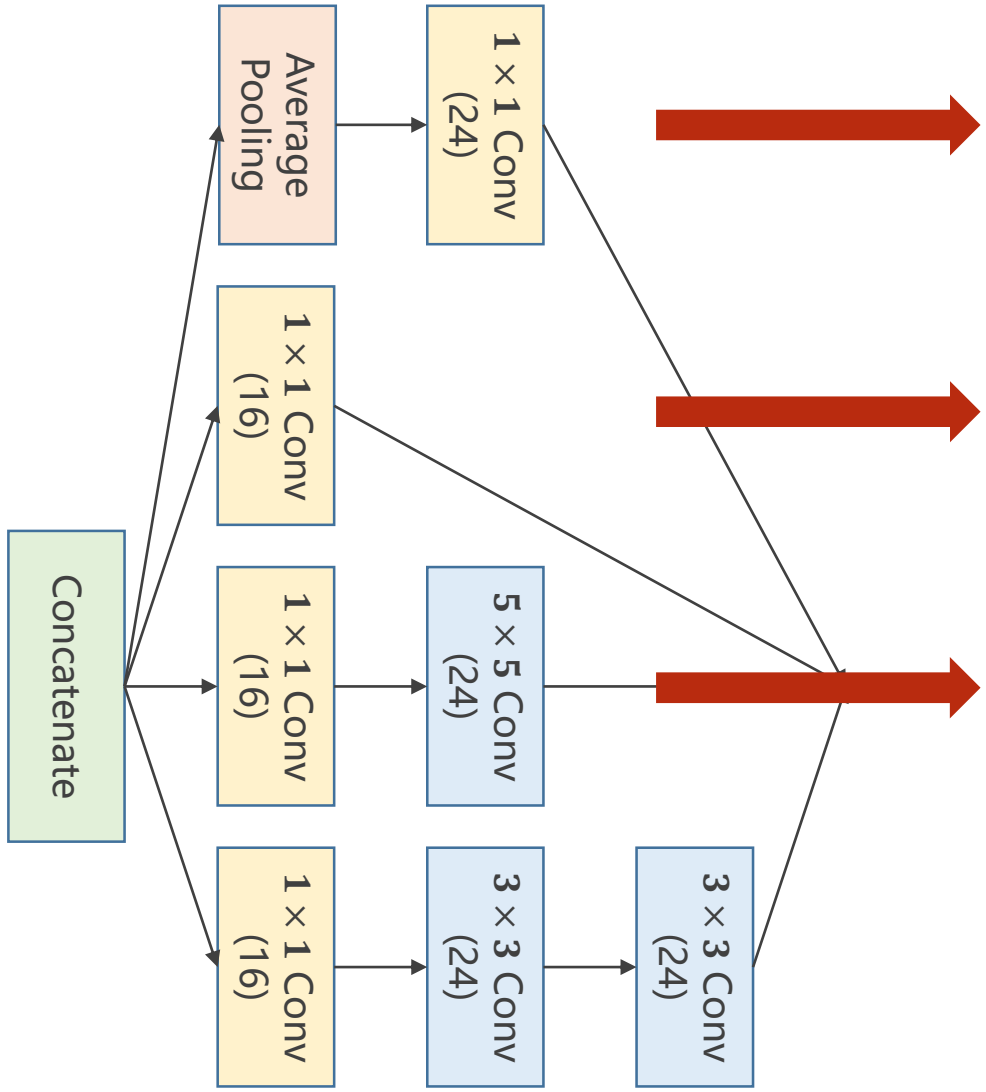
Implementation of Inception Module




```
self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
branch_pool = self.branch_pool(branch_pool)
```

```
self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)

branch1x1 = self.branch1x1(x)
```

Implementation of Inception Module

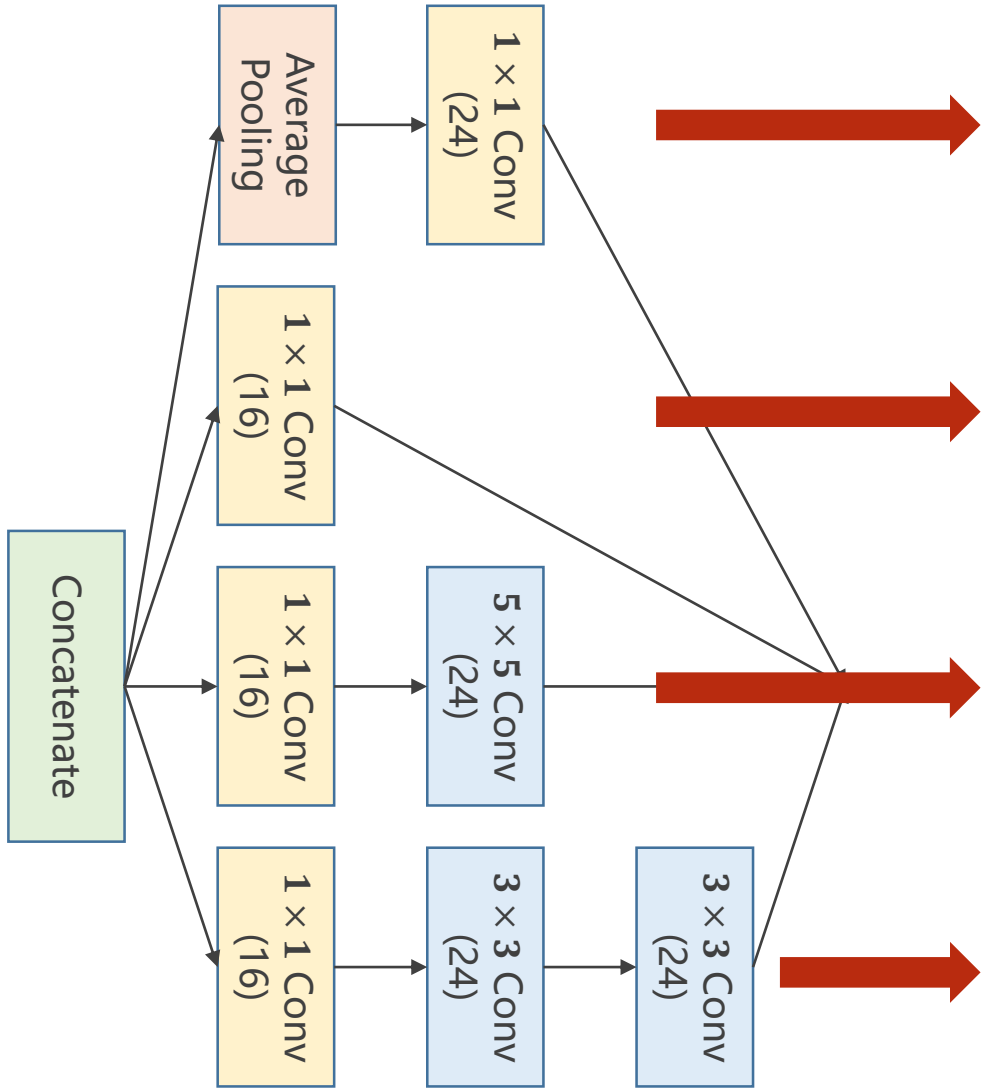



```
self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
branch_pool = self.branch_pool(branch_pool)
```

```
self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)

branch1x1 = self.branch1x1(x)
```

```
self.branch5x5_1 = nn.Conv2d(in_channels,16, kernel_size=1)
self.branch5x5_2 = nn.Conv2d(16, 24, kernel_size=5, padding=2)

branch5x5 = self.branch5x5_1(x)
branch5x5 = self.branch5x5_2(branch5x5)
```

Implementation of Inception Module



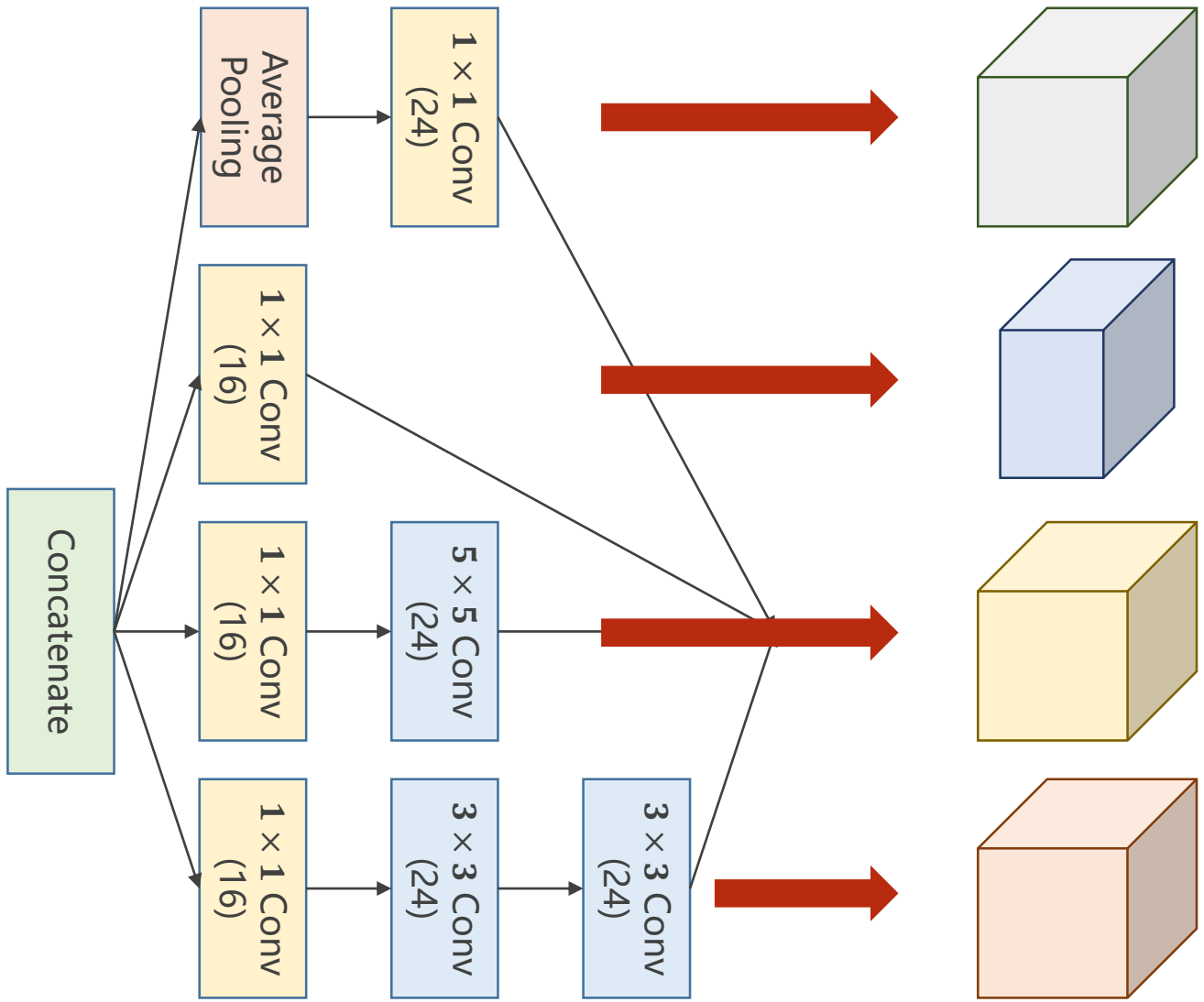

```
self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
branch_pool = self.branch_pool(branch_pool)
```

```
self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)

branch1x1 = self.branch1x1(x)
```

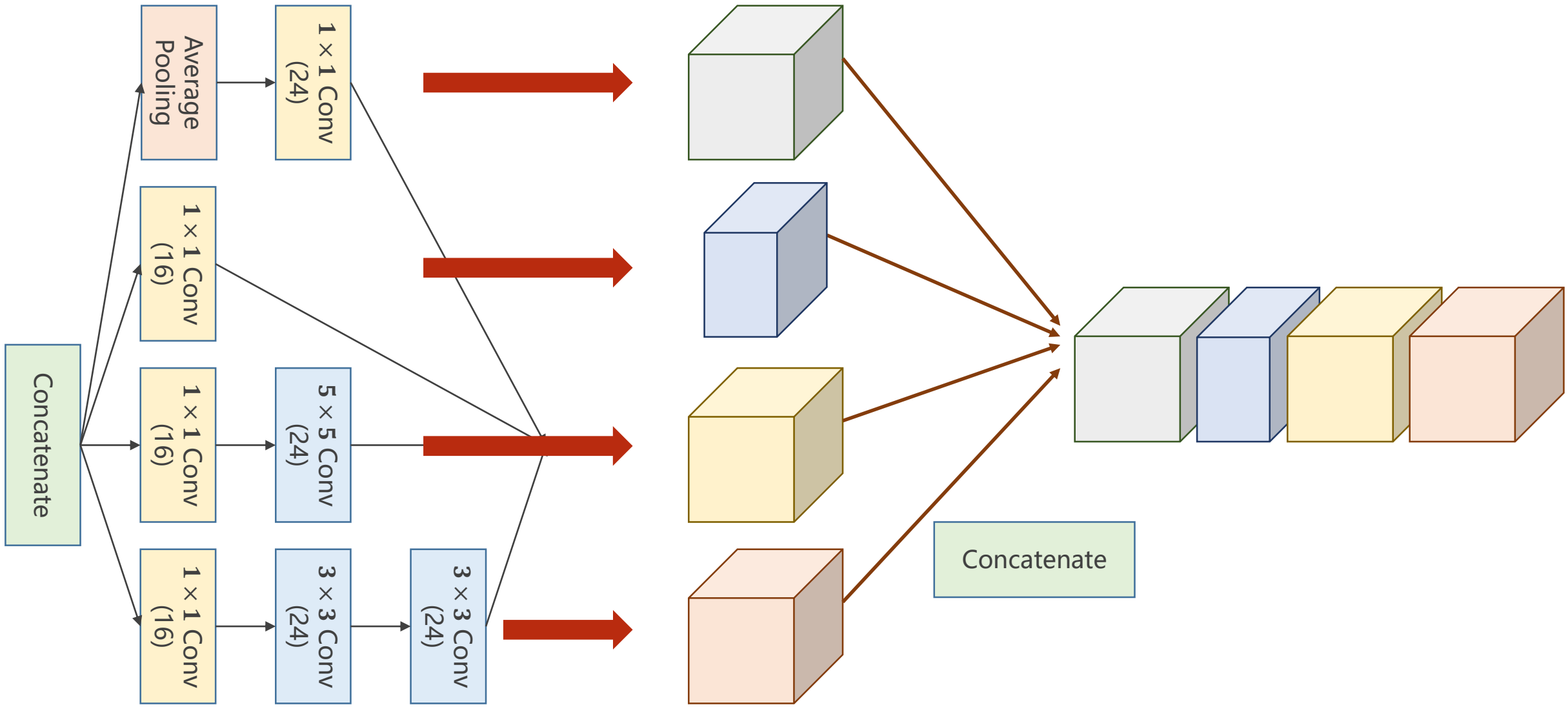
```
self.branch5x5_1 = nn.Conv2d(in_channels,16, kernel_size=1)
self.branch5x5_2 = nn.Conv2d(16, 24, kernel_size=5, padding=2)

branch5x5 = self.branch5x5_1(x)
branch5x5 = self.branch5x5_2(branch5x5)
```

```
self.branch3x3_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
self.branch3x3_2 = nn.Conv2d(16, 24, kernel_size=3, padding=1)
self.branch3x3_3 = nn.Conv2d(24, 24, kernel_size=3, padding=1)

branch3x3 = self.branch3x3_1(x)
branch3x3 = self.branch3x3_2(branch3x3)
```

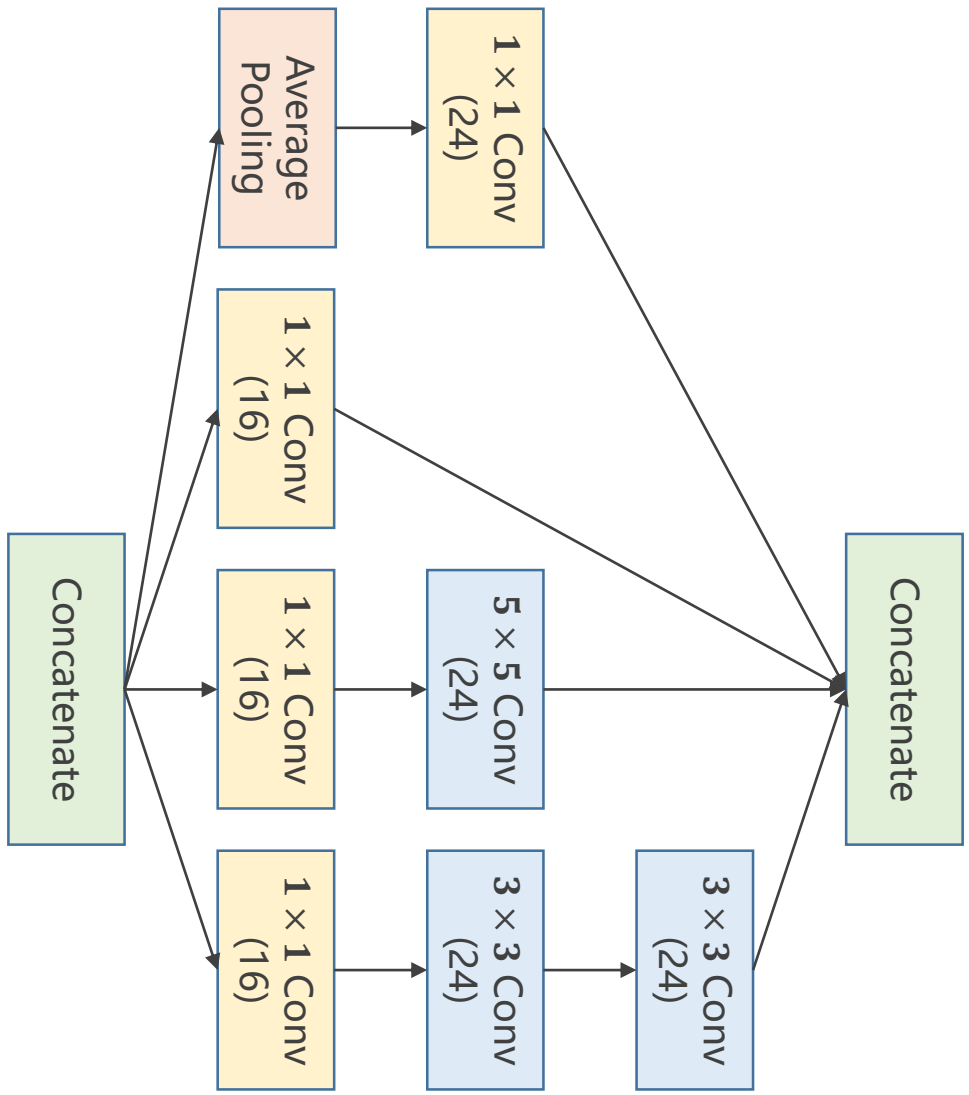
Implementation of Inception Module



Implementation of Inception Module



Implementation of Inception Module




```
outputs = [branch1x1, branch5x5, branch3x3, branch_pool]
return torch.cat(outputs, dim=1)
```

Implementation of Inception Module

```
class InceptionA(nn.Module):
    def __init__(self, in_channels):
        super(InceptionA, self).__init__()
        self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)

        self.branch5x5_1 = nn.Conv2d(in_channels,16, kernel_size=1)
        self.branch5x5_2 = nn.Conv2d(16, 24, kernel_size=5, padding=2)

        self.branch3x3_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
        self.branch3x3_2 = nn.Conv2d(16, 24, kernel_size=3, padding=1)
        self.branch3x3_3 = nn.Conv2d(24, 24, kernel_size=3, padding=1)

        self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

    def forward(self, x):
        branch1x1 = self.branch1x1(x)

        branch5x5 = self.branch5x5_1(x)
        branch5x5 = self.branch5x5_2(branch5x5)

        branch3x3 = self.branch3x3_1(x)
        branch3x3 = self.branch3x3_2(branch3x3)
        branch3x3 = self.branch3x3_3(branch3x3)

        branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
        branch_pool = self.branch_pool(branch_pool)

        outputs = [branch1x1, branch5x5, branch3x3, branch_pool]
        return torch.cat(outputs, dim=1)
```

Using Inception Module

```python
class InceptionA(nn.Module):
    def __init__(self, in_channels):
        super(InceptionA, self).__init__()
        self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)

        self.branch5x5_1 = nn.Conv2d(in_channels,16, kernel_size=1)
        self.branch5x5_2 = nn.Conv2d(16, 24, kernel_size=5, padding=2)

        self.branch3x3_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
        self.branch3x3_2 = nn.Conv2d(16, 24, kernel_size=3, padding=1)
        self.branch3x3_3 = nn.Conv2d(24, 24, kernel_size=3, padding=1)

        self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

    def forward(self, x):
        branch1x1 = self.branch1x1(x)

        branch5x5 = self.branch5x5_1(x)
        branch5x5 = self.branch5x5_2(branch5x5)

        branch3x3 = self.branch3x3_1(x)
        branch3x3 = self.branch3x3_2(branch3x3)
        branch3x3 = self.branch3x3_3(branch3x3)

        branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
        branch_pool = self.branch_pool(branch_pool)

        outputs = [branch1x1, branch5x5, branch3x3, branch_pool]
        return torch.cat(outputs, dim=1)
```

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(88, 20, kernel_size=5)

        self.incep1 = InceptionA(in_channels=10)
        self.incep2 = InceptionA(in_channels=20)

        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(1408, 10)

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = self.incep1(x)
        x = F.relu(self.mp(self.conv2(x)))
        x = self.incep2(x)
        x = x.view(in_size, -1)
        x = self.fc(x)
        return x
```

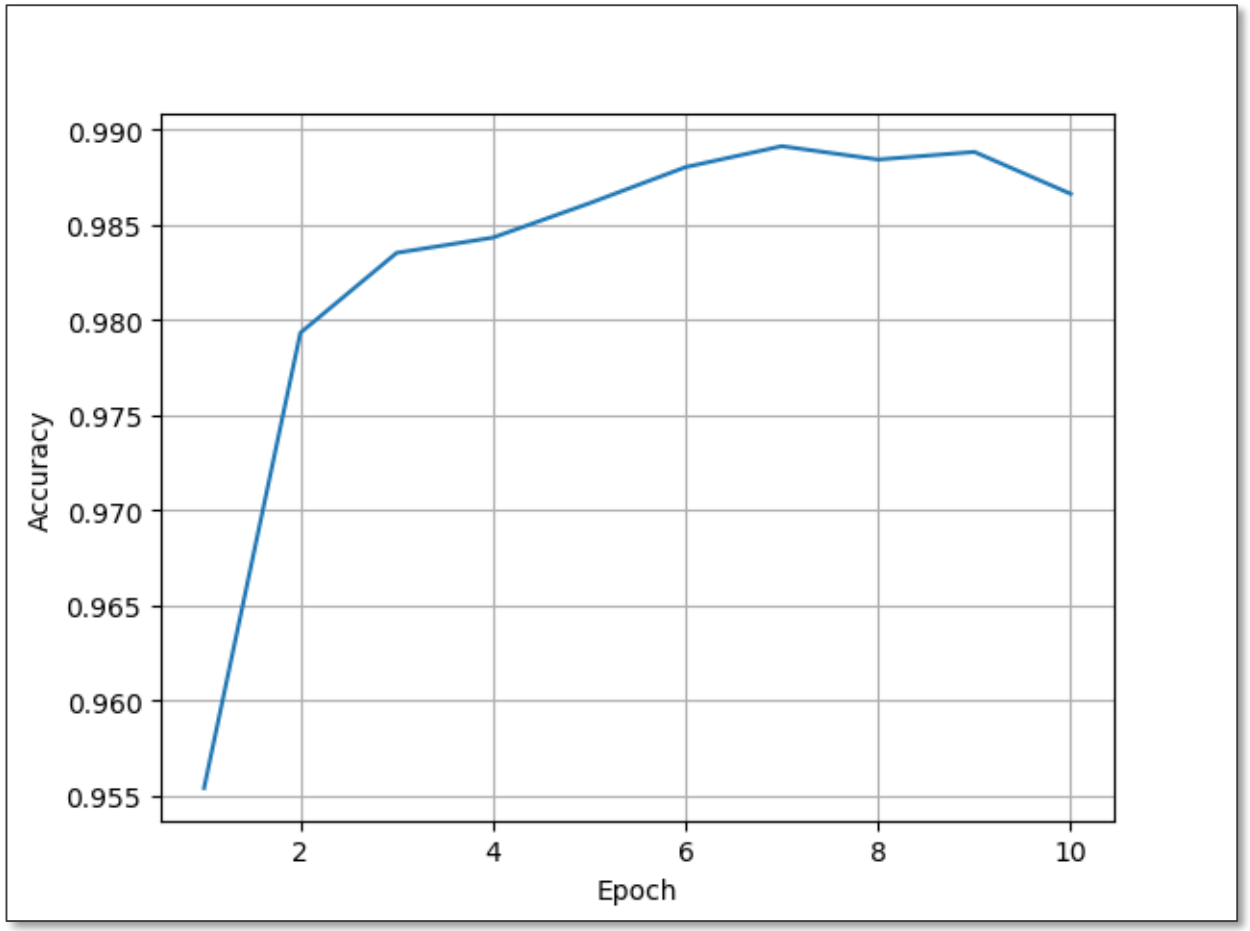
Using Inception Module

```python
class InceptionA(nn.Module):
    def __init__(self, in_channels):
        super(InceptionA, self).__init__()
        self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)

        self.branch5x5_1 = nn.Conv2d(in_channels,16, kernel_size=1)
        self.branch5x5_2 = nn.Conv2d(16, 24, kernel_size=5, padding=2)

        self.branch3x3_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
        self.branch3x3_2 = nn.Conv2d(16, 24, kernel_size=3, padding=1)
        self.branch3x3_3 = nn.Conv2d(24, 24, kernel_size=3, padding=1)

        self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

    def forward(self, x):
        branch1x1 = self.branch1x1(x)

        branch5x5 = self.branch5x5_1(x)
        branch5x5 = self.branch5x5_2(branch5x5)

        branch3x3 = self.branch3x3_1(x)
        branch3x3 = self.branch3x3_2(branch3x3)
        branch3x3 = self.branch3x3_3(branch3x3)

        branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
        branch_pool = self.branch_pool(branch_pool)

        outputs = [branch1x1, branch5x5, branch3x3, branch_pool]
        return torch.cat(outputs, dim=1)
```

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(88, 20, kernel_size=5)

        self.incep1 = InceptionA(in_channels=10)
        self.incep2 = InceptionA(in_channels=20)

        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(1408, 10)

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = self.incep1(x)
        x = F.relu(self.mp(self.conv2(x)))
        x = self.incep2(x)
        x = x.view(in_size, -1)
        x = self.fc(x)
        return x
```

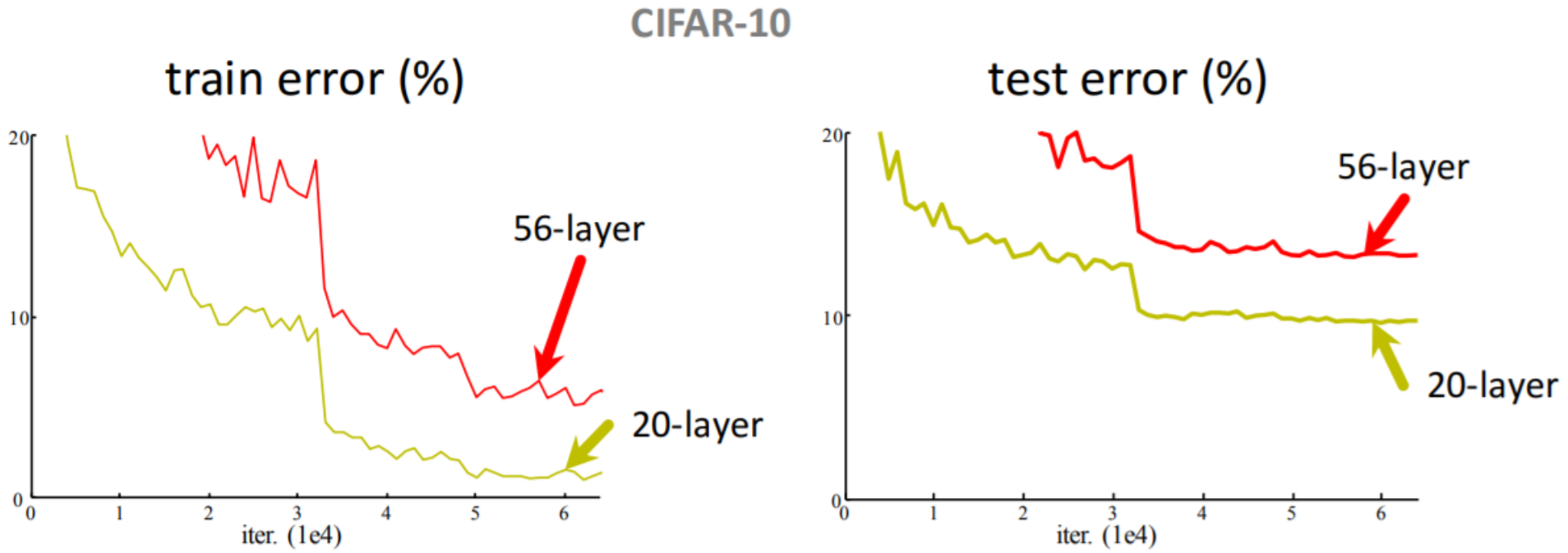
Results of using Inception Module

```
Accuracy on test set: 9 % [982/10000]
[1,   300] loss: 0.141
[1,   600] loss: 0.031
[1,   900] loss: 0.020
Accuracy on test set: 95 % [9554/10000]
[2,   300] loss: 0.015
[2,   600] loss: 0.014
[2,   900] loss: 0.012
Accuracy on test set: 97 % [9793/10000]
……
[9,   300] loss: 0.005
[9,   600] loss: 0.005
[9,   900] loss: 0.005
Accuracy on test set: 98 % [9888/10000]
[10,   300] loss: 0.005
[10,   600] loss: 0.005
[10,   900] loss: 0.005
Accuracy on test set: 98 % [9866/10000]
```



Go Deeper

Can we stack layers to go deeper?

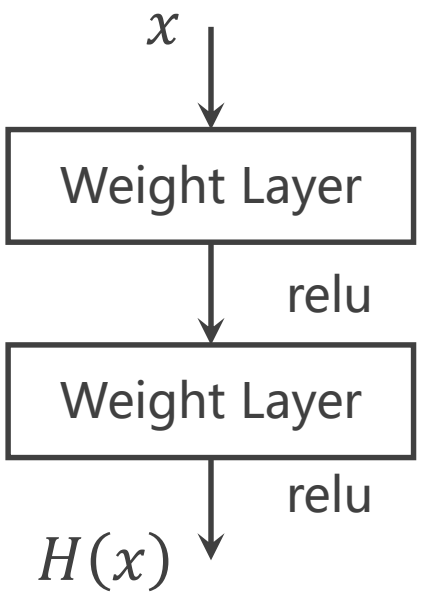


Plain nets: stacking 3x3 conv layers

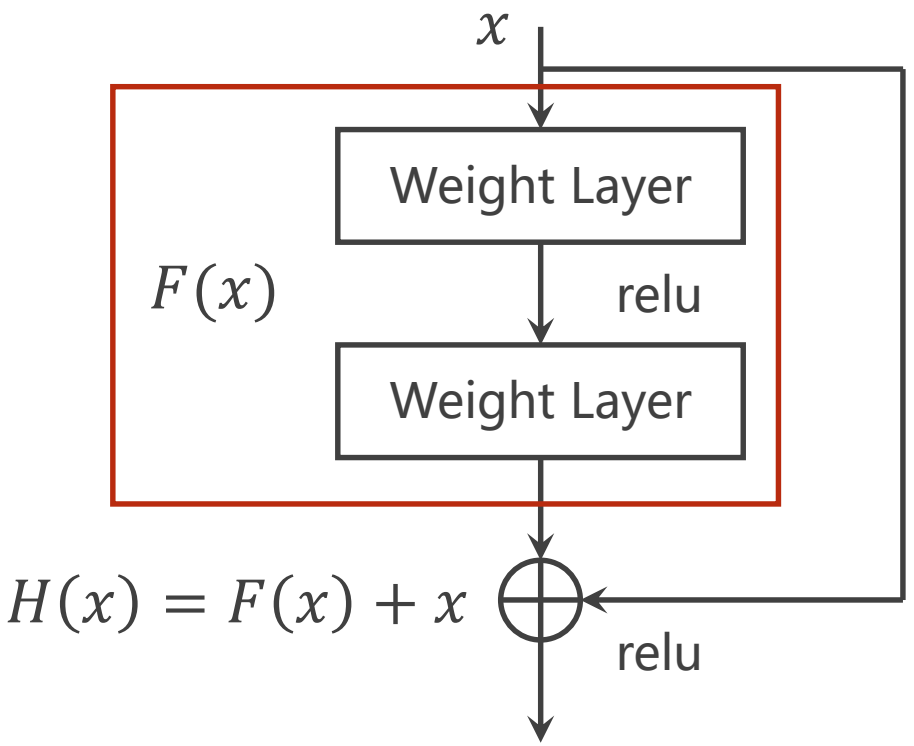
He K, Zhang X, Ren S, et al. Deep Residual Learning for Image Recognition[C]// IEEE Conference on Computer Vision and Pattern Recognition. IEEE Computer Society, 2016:770-778.

Deep Residual Learning

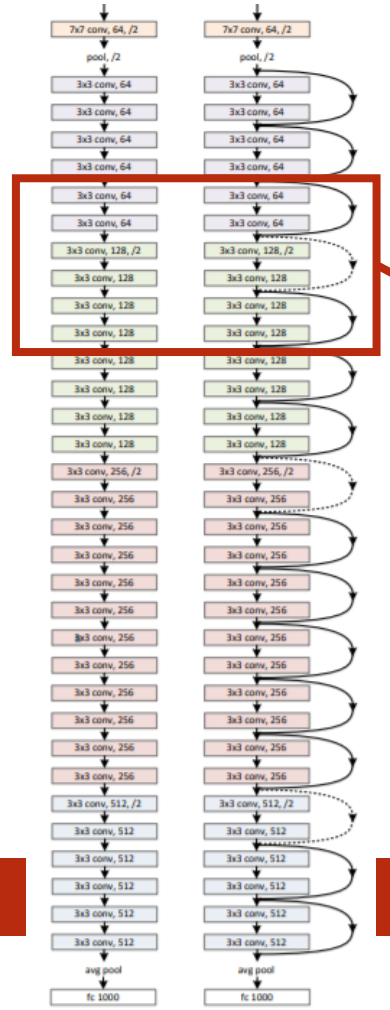
Plain net



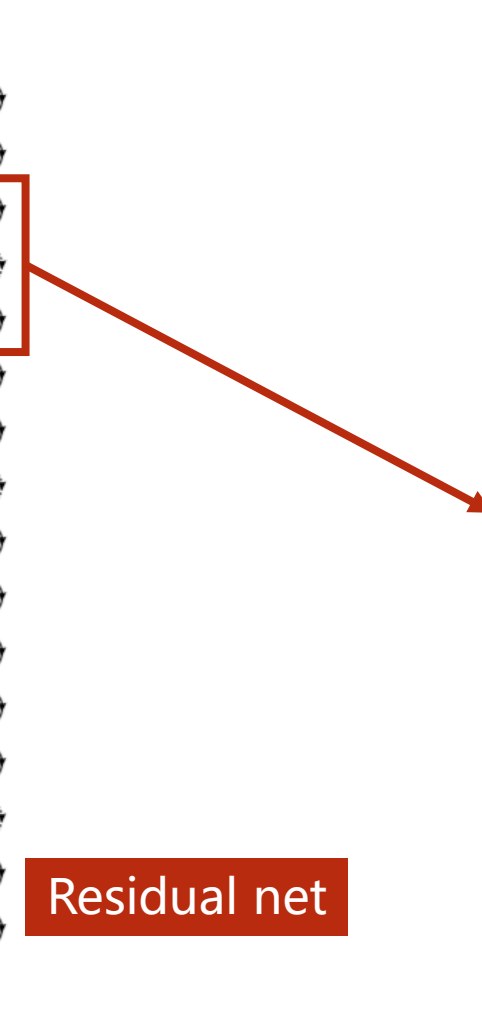
Residual net



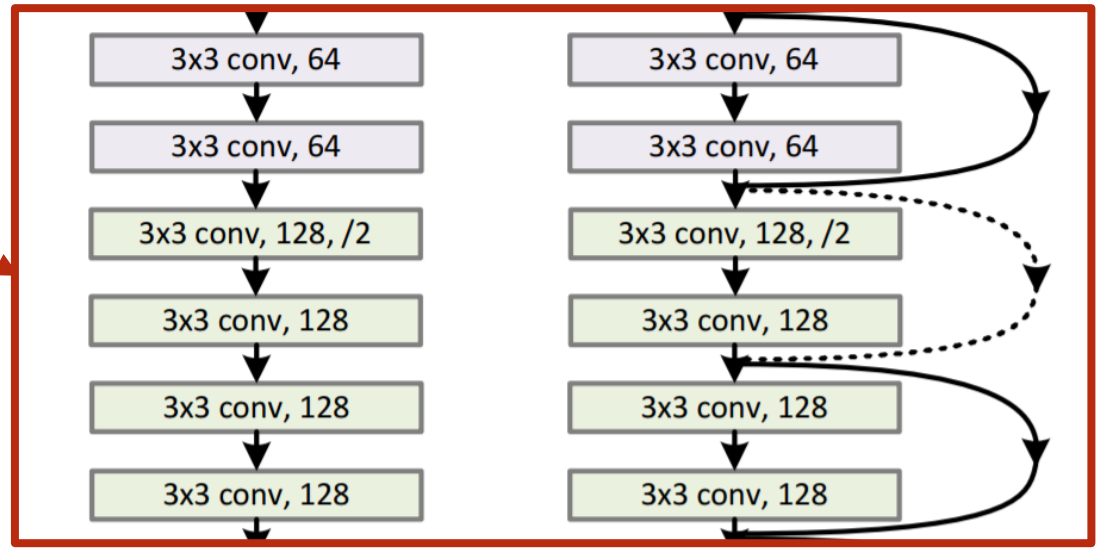
Residual Network



Plain net



Residual net



Implementation of Simple Residual Network

(*batch*, 1,28,28)

(*batch*, 1,28,28)

(*batch*, 16,24,24)

(*batch*, 16,24,24)

(*batch*, 16,12,12)

(*batch*, 16,12,12)

(*batch*, 32,8,8)

(*batch*, 32,8,8)

(*batch*, 32,4,4)

(*batch*, 32,4,4) → (*batch*, 512)

(*batch*, 10)

Input Layer

Conv2d Layer

ReLU Layer

Pooling Layer

Linear Layer

Residual Block

Output Layer

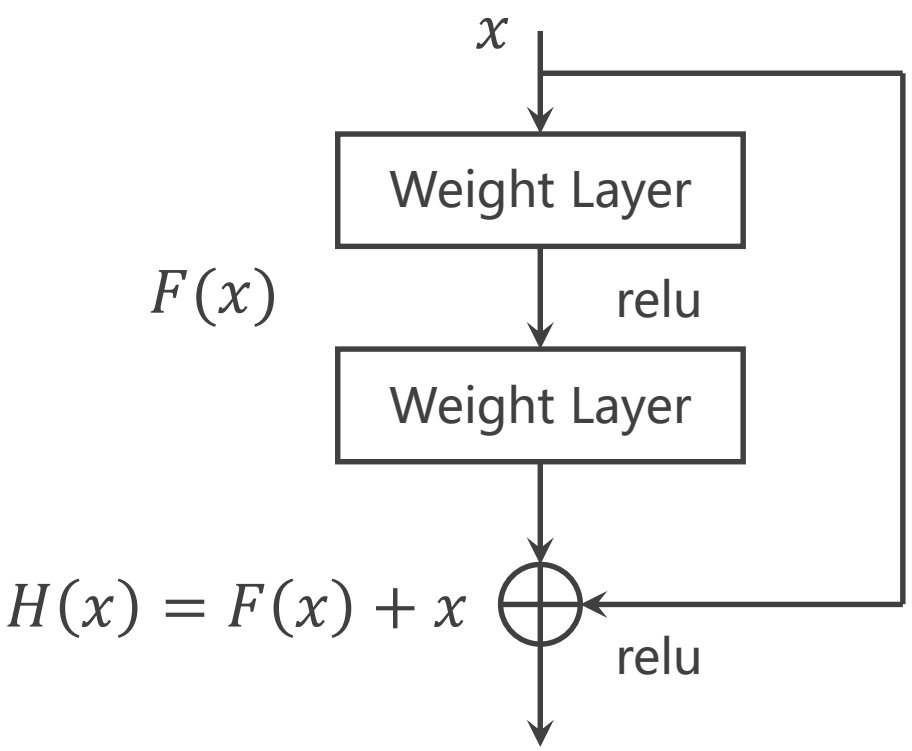
```
self.conv1 = nn.Conv2d(1, 16, kernel_size=5)
```

?

```
self.conv2 = nn.Conv2d(16, 32, kernel_size=5)
```

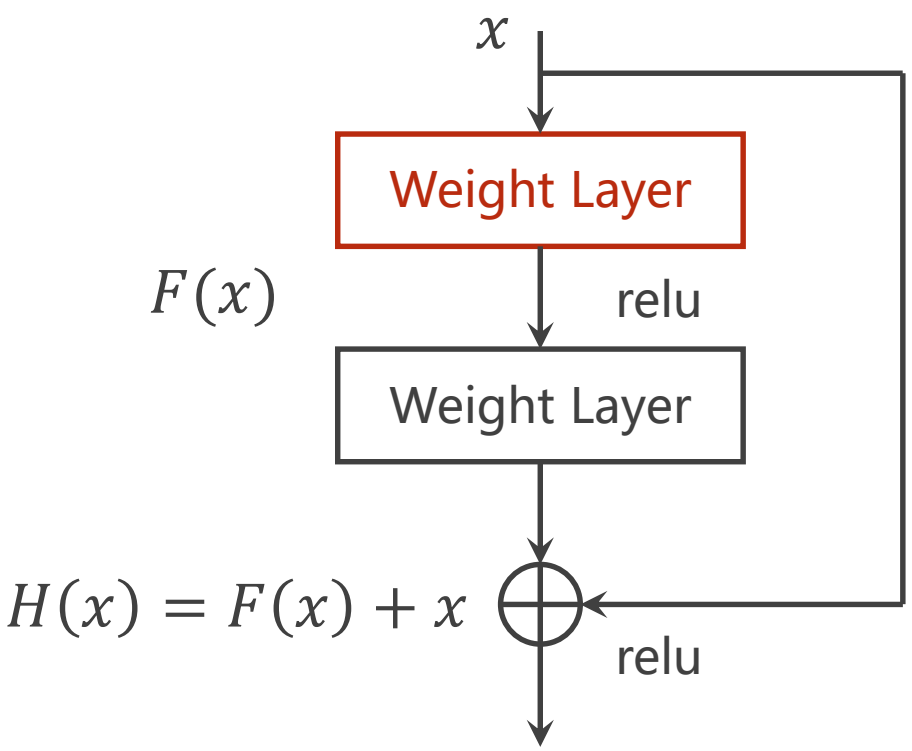
?

Implementation of Residual Block



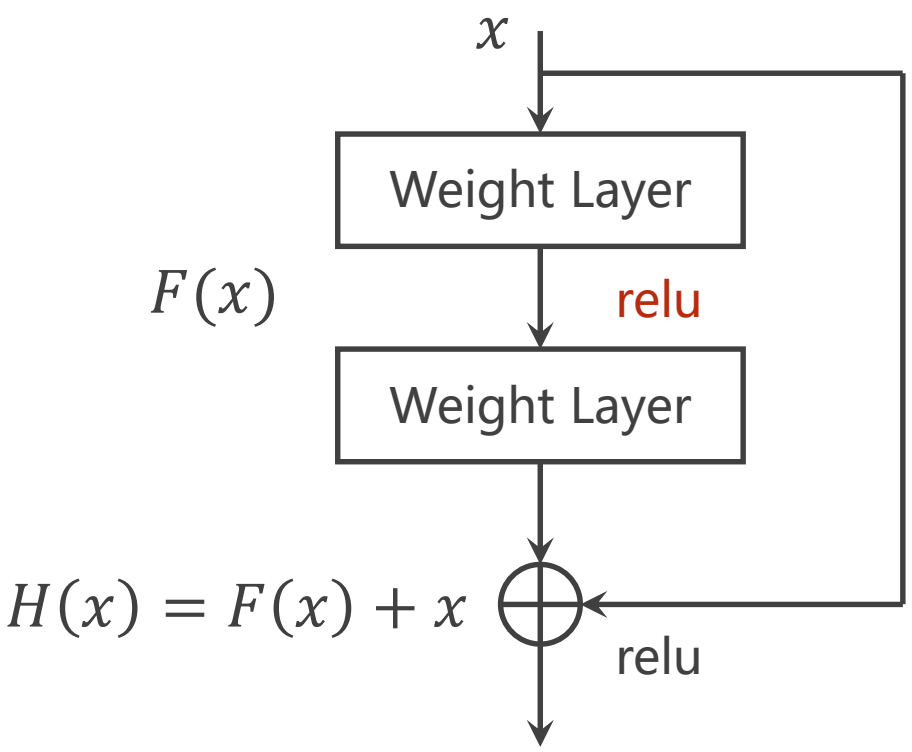

```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.channels = channels
        self.conv1 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)

    def forward(self, x):
        y = F.relu(self.conv1(x))
        y = self.conv2(y)
        return F.relu(x + y)
```

Implementation of Residual Block



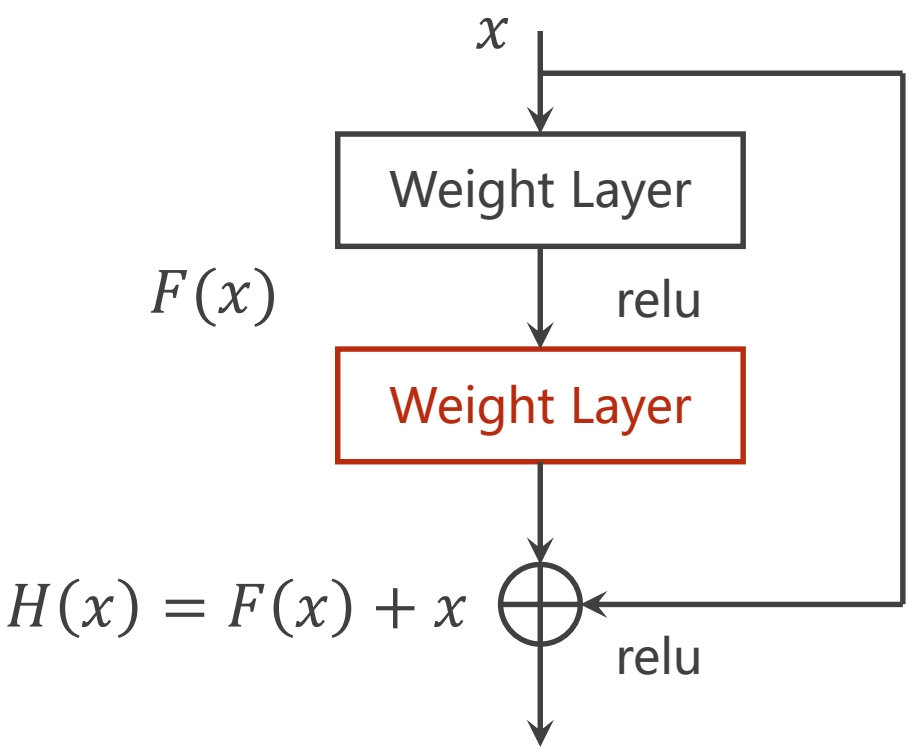

```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.channels = channels
        self.conv1 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)

    def forward(self, x):
        y = F.relu(self.conv1(x))
        y = self.conv2(y)
        return F.relu(x + y)
```

Implementation of Residual Block



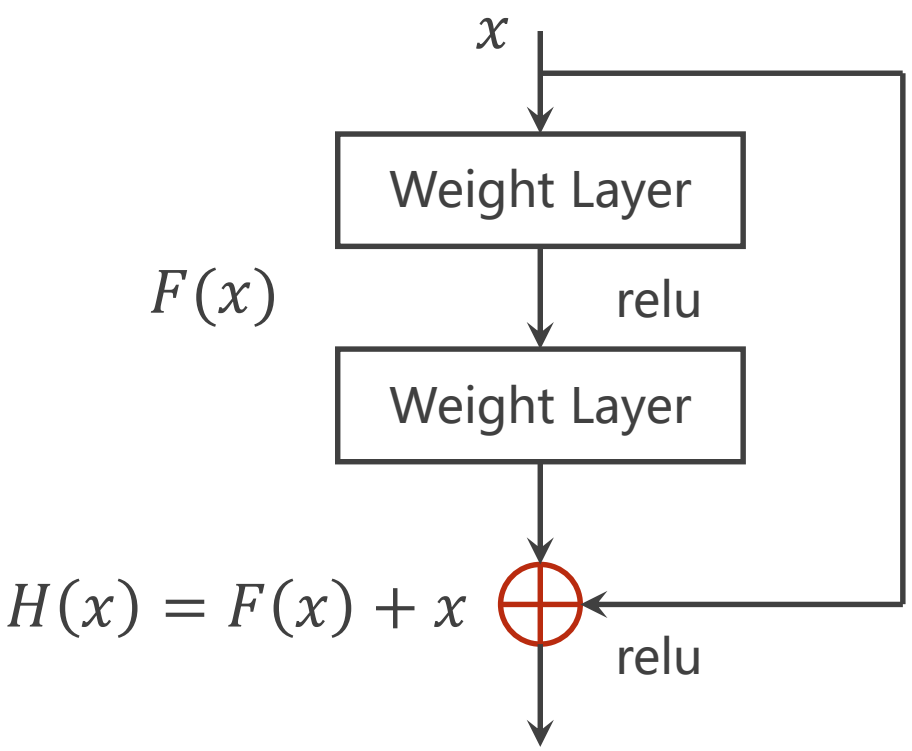

```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.channels = channels
        self.conv1 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)

    def forward(self, x):
        y = F.relu(self.conv1(x))
        y = self.conv2(y)
        return F.relu(x + y)
```

Implementation of Residual Block



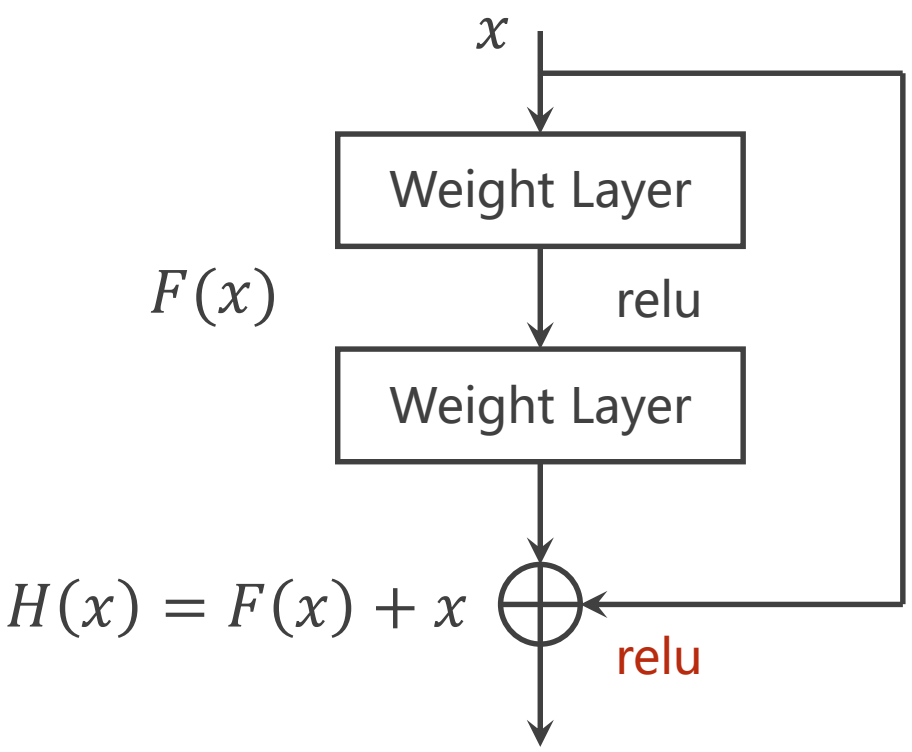

```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.channels = channels
        self.conv1 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)

    def forward(self, x):
        y = F.relu(self.conv1(x))
        y = self.conv2(y)
        return F.relu(x + y)
```

Implementation of Residual Block



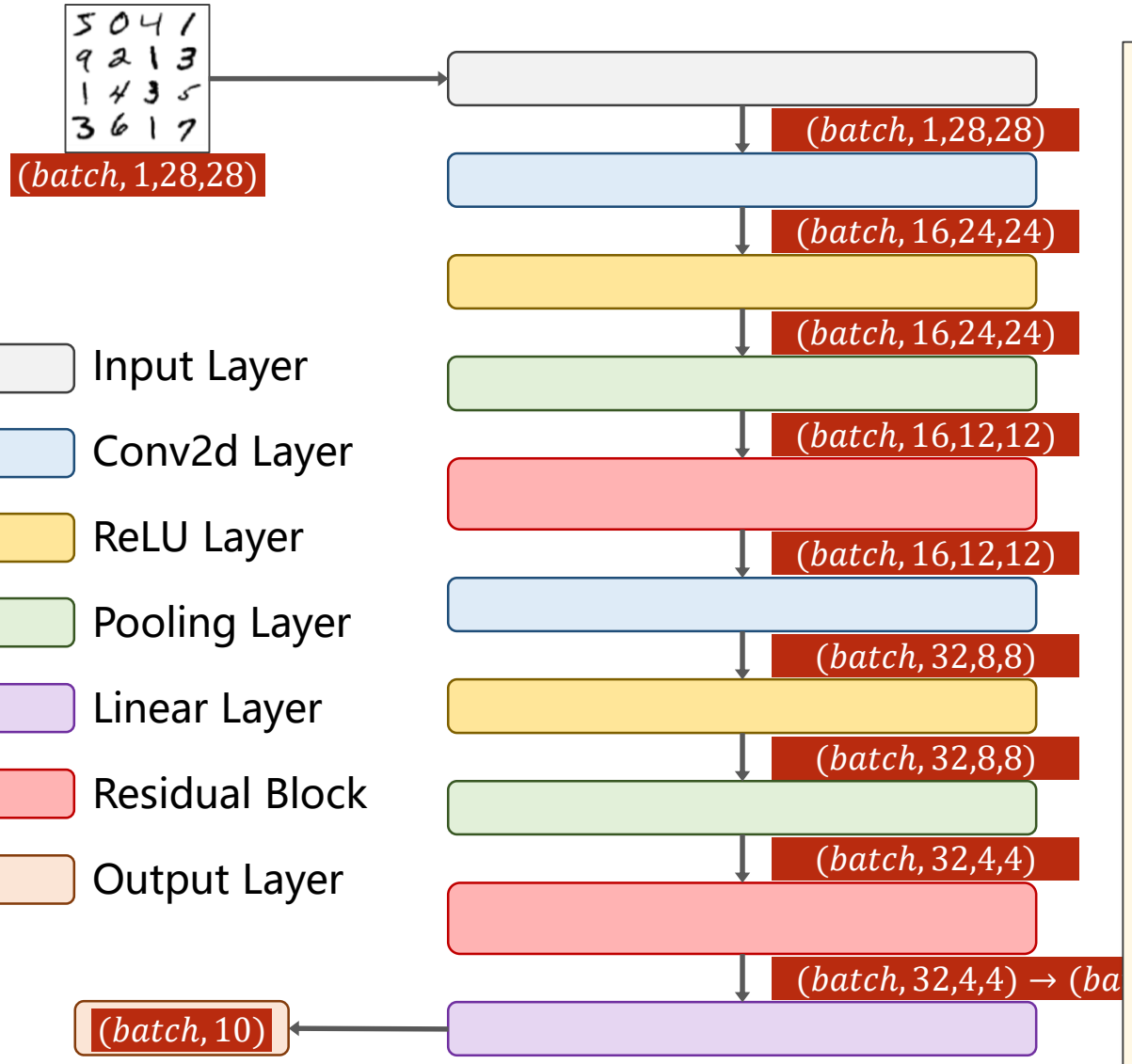

```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.channels = channels
        self.conv1 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)

    def forward(self, x):
        y = F.relu(self.conv1(x))
        y = self.conv2(y)
        return F.relu(x + y)
```

Implementation of Residual Block



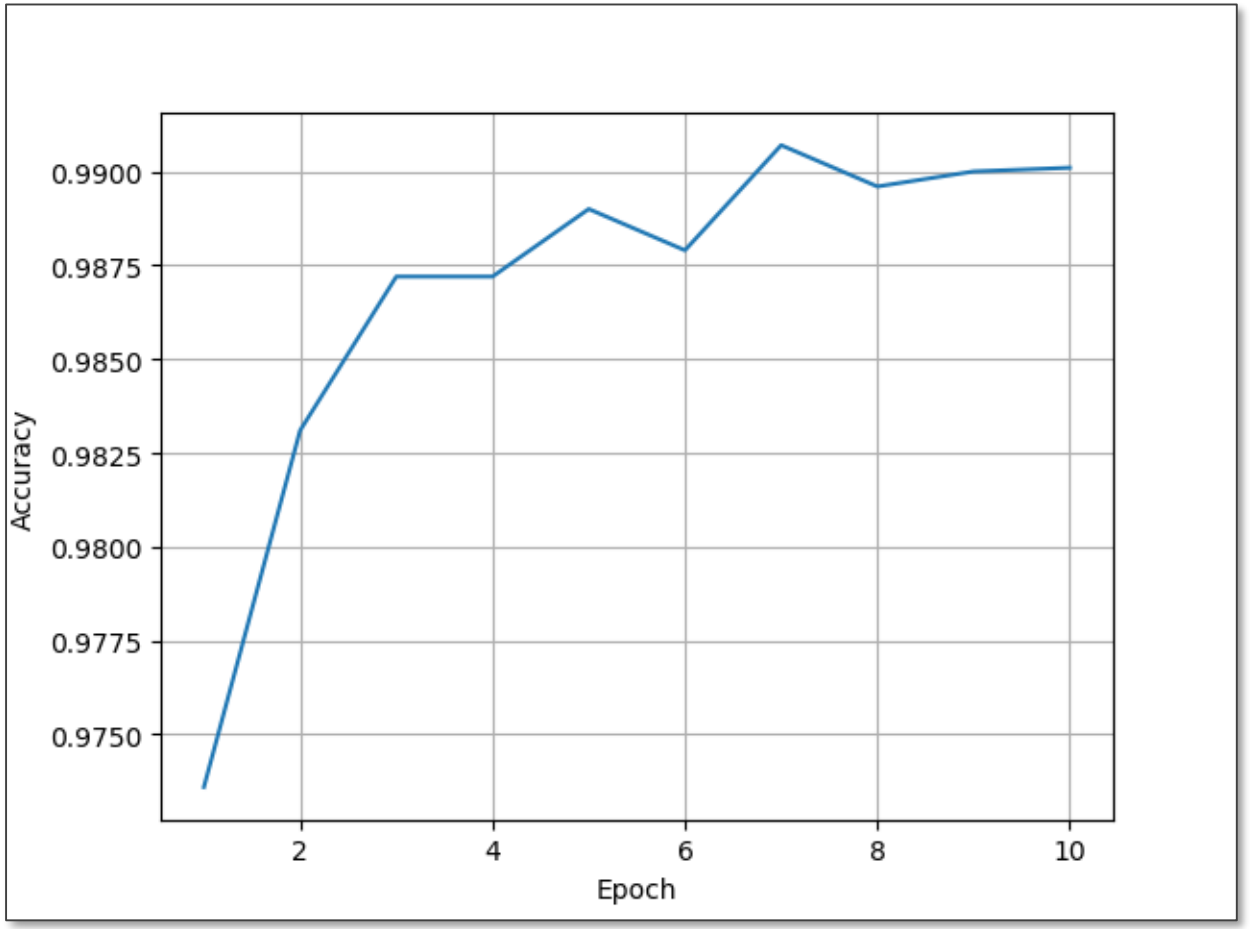

```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.channels = channels
        self.conv1 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)

    def forward(self, x):
        y = F.relu(self.conv1(x))
        y = self.conv2(y)
        return F.relu(x + y)
```

Implementation of Simple Residual Network

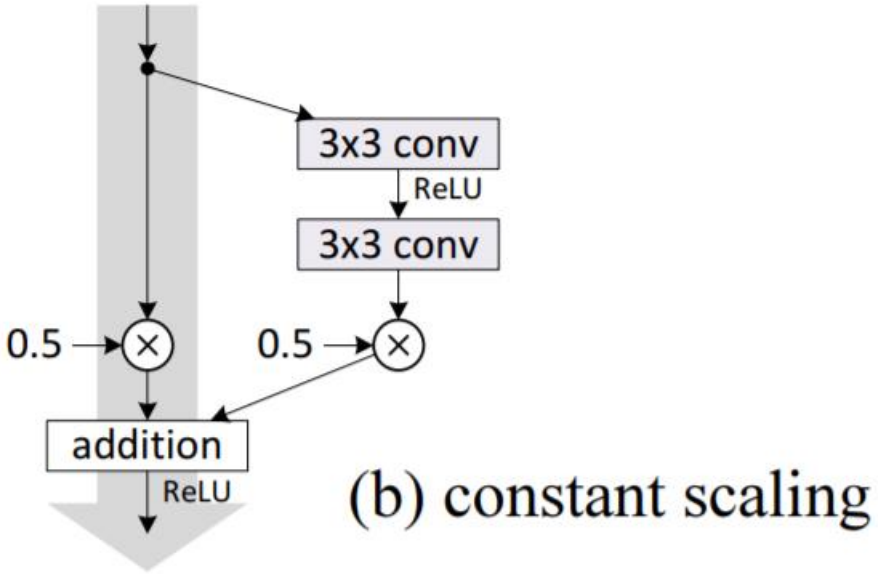



```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=5)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5)
        self.mp = nn.MaxPool2d(2)

        self.rblock1 = ResidualBlock(16)
        self.rblock2 = ResidualBlock(32)

        self.fc = nn.Linear(512, 10)

    def forward(self, x):
        in_size = x.size(0)
        x = self.mp(F.relu(self.conv1(x)))
        x = self.rblock1(x)
        x = self.mp(F.relu(self.conv2(x)))
        x = self.rblock2(x)
        x = x.view(in_size, -1)
        x = self.fc(x)
        return x
```

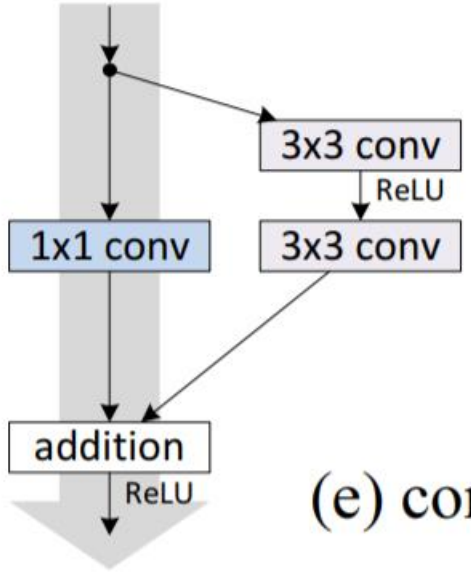
```
Accuracy on test set: 9 % [916/10000]
[1,   300] loss: 0.074
[1,   600] loss: 0.021
[1,   900] loss: 0.017
Accuracy on test set: 97 % [9736/10000]
[2,   300] loss: 0.013
[2,   600] loss: 0.011
[2,   900] loss: 0.011
Accuracy on test set: 98 % [9831/10000]
......
[9,   300] loss: 0.003
[9,   600] loss: 0.004
[9,   900] loss: 0.004
Accuracy on test set: 99 % [9900/10000]
[10,   300] loss: 0.003
[10,   600] loss: 0.003
[10,   900] loss: 0.004
Accuracy on test set: 99 % [9901/10000]
```



Exercise 11-1: Reading Paper and Implementing



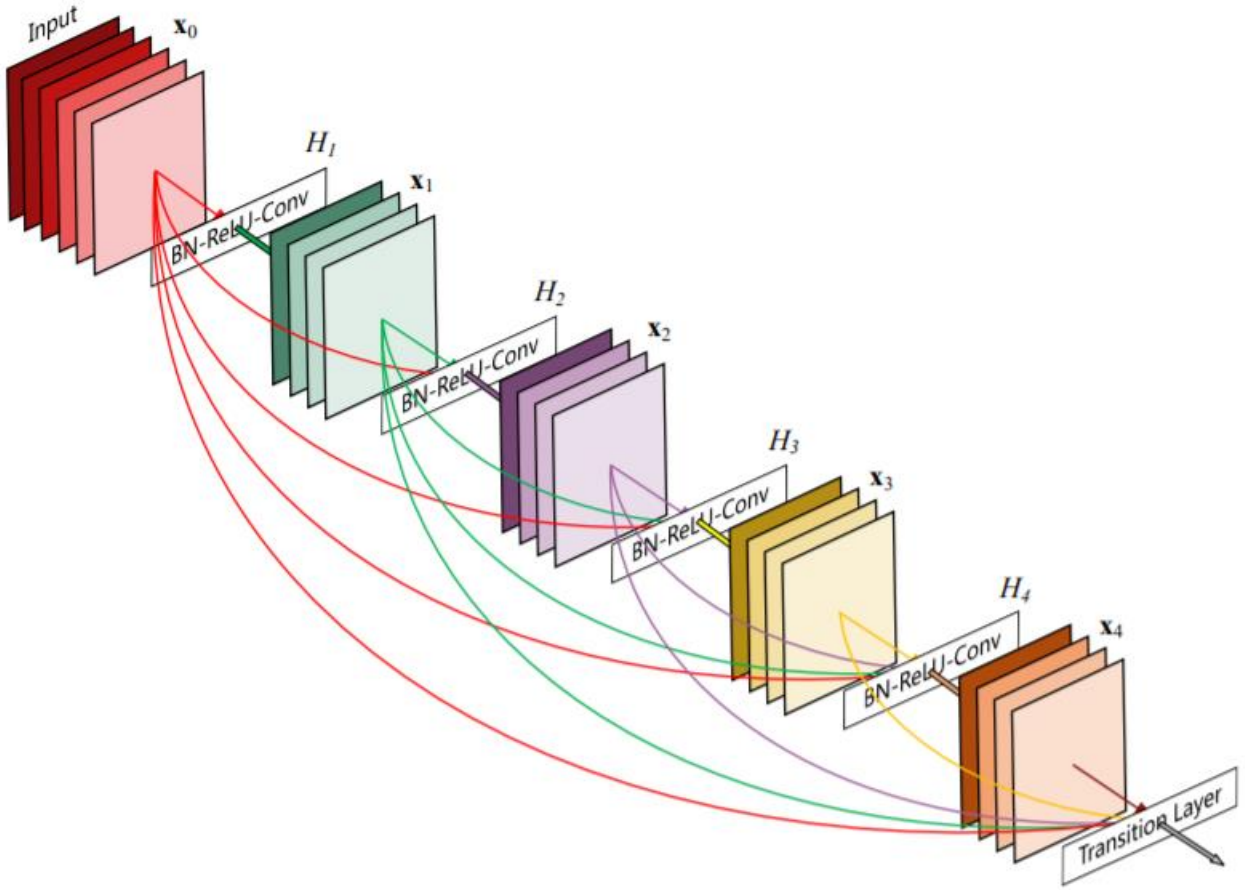
(b) constant scaling



(e) conv shortcut

He K, Zhang X, Ren S, et al. Identity Mappings in Deep Residual Networks[C]

Exercise 11-2: Reading and Implementing DenseNet



Huang G, Liu Z, Laurens V D M, et al. Densely Connected Convolutional Networks[J]. 2016:2261-2269.

PyTorch Tutorial

11. Advanced CNN