



教育部产学合作协同育人项目资助  
2022年北京交通大学《深度学习》课程

# 实验3 卷积神经网络实验

主讲教师：丛润民  
助 教：盛荣晖





## 1. 卷积神经网络

- 卷积基本操作
- 手动实现卷积层
- 图像分类示例

## 2. 空洞卷积

- 空洞卷积的概念
- 优点与适用性
- 缺点与解决方法
- 使用PyTorch实现与实验结果

## 3. 残差网络

- 解决的问题
- 网络结构
- 使用PyTorch实现与实验结果

## 4. 实验要求

- 数据集介绍
- 卷积实验
- 空洞卷积实验
- 残差网络实验



## 卷积神经网络 (CNN)

- 一维卷积

用于时序数据

- 二维卷积

用于网格数据 (如图片)

- 三维卷积

用于时空数据

## 卷积神经网络基本概念

- 卷积运算

- 填充

- 步幅

- 多通道输入和输出

- 池化



# 卷积运算

输入

|   |   |   |
|---|---|---|
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

核

|   |   |
|---|---|
| 0 | 1 |
| 2 | 3 |

=

|    |    |
|----|----|
| 19 | 25 |
| 37 | 43 |

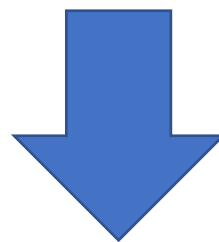
$$\begin{aligned}0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 &= 19, \\1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 &= 25, \\3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 &= 37, \\4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 &= 43.\end{aligned}$$

原始输入维度( $H, W$ ) = (3,3)

卷积核维度( $k_h, k_w$ ) = (2,2)

输出维度( $y_h, y_w$ ) = (2,2)

步长为1



$$y_h = H - k_h + 1$$

$$y_w = W - k_w + 1$$

卷积层的输出形状与输入形状、卷积核窗口有关



# 卷积运算

## 定义卷积运算

```
1 import torch
2 from torch import nn

executed in 5ms, finished 22:17:04 2020-06-16

1 def corr2d(X,K):
2     """
3     X: 输入, shape (H,W)
4     K: 卷积核, shape (k_h,k_w)
5     """
6     H, W = X.shape
7     k_h, k_w = K.shape
8     #初始化结果矩阵
9     Y = torch.zeros((H - k_h + 1, W - k_w + 1))
10    for i in range(Y.shape[0]):
11        for j in range(Y.shape[1]):
12            Y[i, j] = (X[i: i + k_h, j: j + k_w] * K).sum()
13    return Y
```

$$y_h = H - k_h + 1$$
$$y_w = W - k_w + 1$$

## 示例

```
1 #验证卷积操作
2 X = torch.tensor([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
3 K = torch.tensor([[0, 1], [2, 3]])
4 corr2d(X, K)

executed in 7ms, finished 22:18:39 2020-06-16

tensor([[19., 25.],
        [37., 43.]])
```

| 输入                                                                                                                                   |    | 核 |   | 输出 |   |   |   |   |   |   |                                                                           |   |   |   |   |   |                                                                               |    |    |    |    |
|--------------------------------------------------------------------------------------------------------------------------------------|----|---|---|----|---|---|---|---|---|---|---------------------------------------------------------------------------|---|---|---|---|---|-------------------------------------------------------------------------------|----|----|----|----|
| <table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table> | 0  | 1 | 2 | 3  | 4 | 5 | 6 | 7 | 8 | * | <table><tr><td>0</td><td>1</td></tr><tr><td>2</td><td>3</td></tr></table> | 0 | 1 | 2 | 3 | = | <table><tr><td>19</td><td>25</td></tr><tr><td>37</td><td>43</td></tr></table> | 19 | 25 | 37 | 43 |
| 0                                                                                                                                    | 1  | 2 |   |    |   |   |   |   |   |   |                                                                           |   |   |   |   |   |                                                                               |    |    |    |    |
| 3                                                                                                                                    | 4  | 5 |   |    |   |   |   |   |   |   |                                                                           |   |   |   |   |   |                                                                               |    |    |    |    |
| 6                                                                                                                                    | 7  | 8 |   |    |   |   |   |   |   |   |                                                                           |   |   |   |   |   |                                                                               |    |    |    |    |
| 0                                                                                                                                    | 1  |   |   |    |   |   |   |   |   |   |                                                                           |   |   |   |   |   |                                                                               |    |    |    |    |
| 2                                                                                                                                    | 3  |   |   |    |   |   |   |   |   |   |                                                                           |   |   |   |   |   |                                                                               |    |    |    |    |
| 19                                                                                                                                   | 25 |   |   |    |   |   |   |   |   |   |                                                                           |   |   |   |   |   |                                                                               |    |    |    |    |
| 37                                                                                                                                   | 43 |   |   |    |   |   |   |   |   |   |                                                                           |   |   |   |   |   |                                                                               |    |    |    |    |



## 将卷积运算封装成卷积层

- 卷积层参数：卷积核、偏差（随机初始化）

```
▼ 1 class Conv2D(nn.Module):  
▼ 2     def __init__(self, kernel_size):  
3         super(Conv2D, self).__init__()  
4         #初始化卷积层的2个参数: 卷积核、偏差  
5         self.weight = nn.Parameter(torch.randn(kernel_size))  
6         self.bias = nn.Parameter(torch.randn(1))  
7  
▼ 8     def forward(self, x):  
9         return corr2d(x, self.weight) + self.bias
```



# 卷积的应用

## 卷积用来边缘检测

- 给定一个6\*8的图像X，中间4列为黑（0），其余为白（1）

```
1 X = torch.ones(6, 8)
2 X[:, 2:6] = 0
3 X
```

executed in 9ms, finished 12:16:18 2020-06-17

```
tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])
```

- 给定卷积核 $K = (1, -1)$

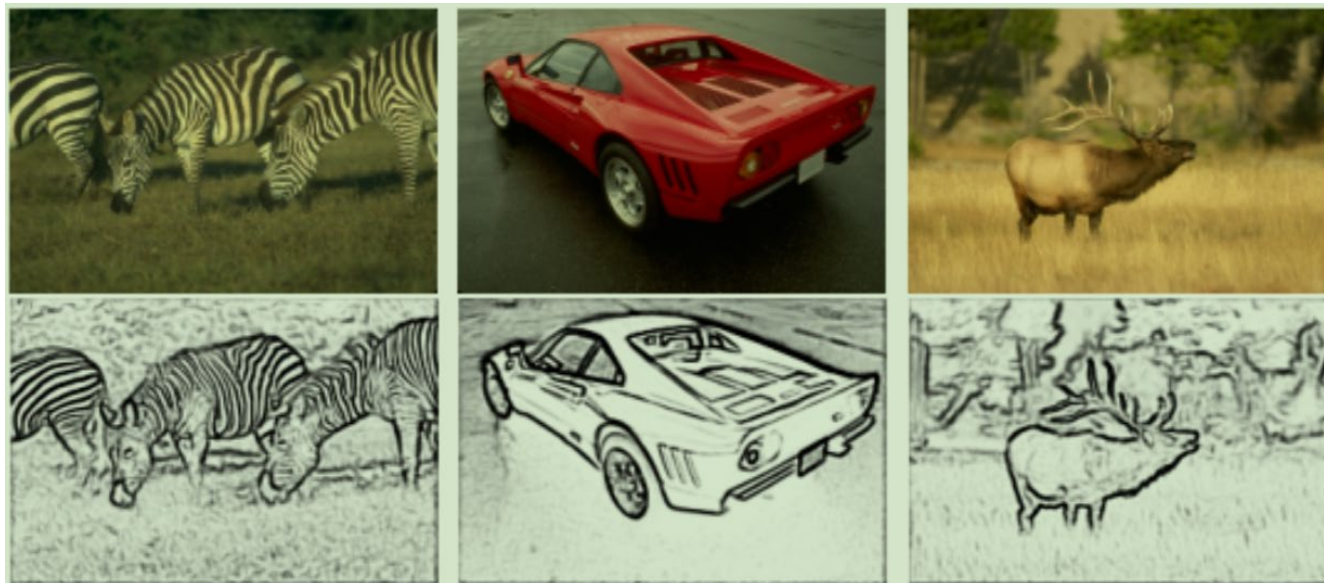
```
1 K = torch.tensor([[1, -1]])
```

executed in 4ms, finished 12:20:37 2020-06-17

```
1 Y = corr2d(X, K)
2 Y
```

executed in 10ms, finished 12:20:43 2020-06-17

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```



检测出边缘的颜色变化



# 反向传播训练卷积核

## 卷积用来边缘检测

- 给定一个6\*8的图像 $X$ ,  
中间4列为黑 (0) , 其余为白 (1)

```
1 X = torch.ones(6, 8)
2 X[:, 2:6] = 0
3 X
```

executed in 9ms, finished 12:16:18 2020-06-17

```
tensor([[1., 1., 0., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0., 0., 0.]])
```

已知输入 $X$ 和真实输出 $Y$

通过反向传播可以学习卷积核和偏差

- 给定卷积核 $K = (1, -1)$

```
1 K = torch.tensor([[1, -1]])
```

executed in 4ms, finished 12:20:37 2020-06-17

```
1 Y = corr2d(X, K)
2 Y
```

executed in 10ms, finished 12:20:43 2020-06-17

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

## 反向传播学习卷积核

```
1 # 构造一个核数组形状是(1, 2)的二维卷积层
2 conv2d = Conv2D(kernel_size=(1, 2))
3
4 step = 20
5 lr = 0.01
6 for i in range(step):
7     Y_hat = conv2d(X)
8     loss = ((Y_hat - Y) ** 2).sum()
9     backward()
10
11     # 学习率下降
12     conv2d.weight.data -= lr * conv2d.weight.grad
13     conv2d.bias.data -= lr * conv2d.bias.grad
14
15     # 梯度清零
16     conv2d.weight.grad.fill_(0)
17     conv2d.bias.grad.fill_(0)
18     (i + 1) % 5 == 0:
19         print('Step %d, loss %.3f' % (i + 1, 1.item()))
```

executed in 427ms, finished 12:26:12 2020-06-17

```
Step 5, loss 0.118
Step 10, loss 0.030
Step 15, loss 0.008
Step 20, loss 0.002
```

- 学习得到的卷积核

```
1 print("weight: ", conv2d.weight.data)
2 print("bias: ", conv2d.bias.data)
```

executed in 6ms, finished 12:26:39 2020-06-17

```
weight: tensor([[ 1.0110, -1.0125]])
bias: tensor([0.0008])
```



# 填充和步幅

## 卷积层的2个超参数：填充、步幅

- 填充：通常用“0”填充
- 步幅：通常用来减少输出的高和宽

输入                      核                      输出

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 2 | 0 |
| 0 | 3 | 4 | 5 | 0 |
| 0 | 6 | 7 | 8 | 0 |
| 0 | 0 | 0 | 0 | 0 |

 \* 

|   |   |
|---|---|
| 0 | 1 |
| 2 | 3 |

 = 

|    |    |    |    |
|----|----|----|----|
| 0  | 3  | 8  | 4  |
| 9  | 19 | 25 | 10 |
| 21 | 37 | 43 | 16 |
| 6  | 7  | 8  | 0  |

## 实现填充和步幅

```
1 # 定义一个函数来计算卷积层。它对输入和输出做相应的升维和降维
2 def comp_conv2d(conv2d, X):
3     # (1, 1)代表批量大小和通道数均为1
4     X = X.view((1, 1) + X.shape)
5     Y = conv2d(X)
6     return Y.view(Y.shape[2:]) # 排除不关心的前两维：批量和通道
7
8 # 注意这里是两侧分别填充1行或列，所以在两侧一共填充2行或列
9 conv2d = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3, padding=1, stride=2)
10
11 X = torch.rand(8, 8)
12 comp_conv2d(conv2d, X).shape
```

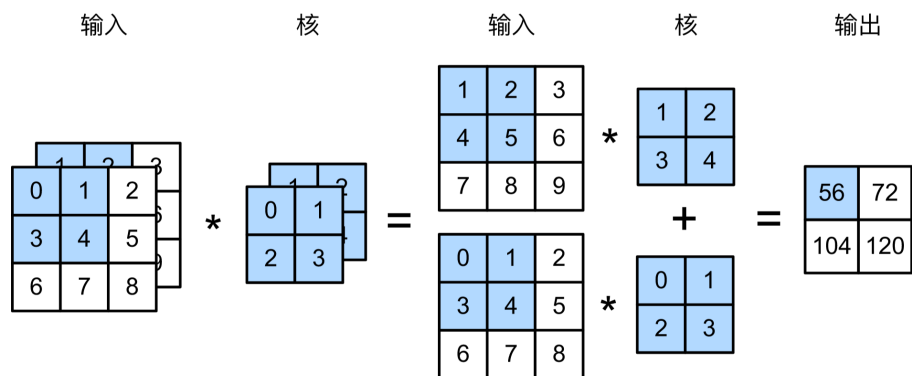
executed in 93ms, finished 15:09:39 2020-06-17

torch.Size([4, 4])



## 多输入通道

例如：彩色图片有红、绿、蓝3个通道，用  $3 \times H \times W$  表示



输入维度:  $(2, H, W)$

卷积核维度:  $(2, k_h, k_w)$

## 实现多输入通道

遍历每个通道做卷积运算，再将不同通道的值相加

```
def corr2d_multi_in(X, K):
    # 输入X: 维度(C_in, H, W)
    # 卷积核K: 维度(C_in, k_h, k_w)
    res = corr2d(X[0, :, :], K[0, :, :])
    for i in range(1, X.shape[0]):
        # 按通道相加
        res += corr2d(X[i, :, :], K[i, :, :])
    return res
```

executed in 4ms, finished 15:45:14 2020-06-17

```
X = torch.tensor([[[0, 1, 2], [3, 4, 5], [6, 7, 8]],
                  [[1, 2, 3], [4, 5, 6], [7, 8, 9]]])
K = torch.tensor([[[0, 1], [2, 3]], [[1, 2], [3, 4]]])
corr2d_multi_in(X, K)
```

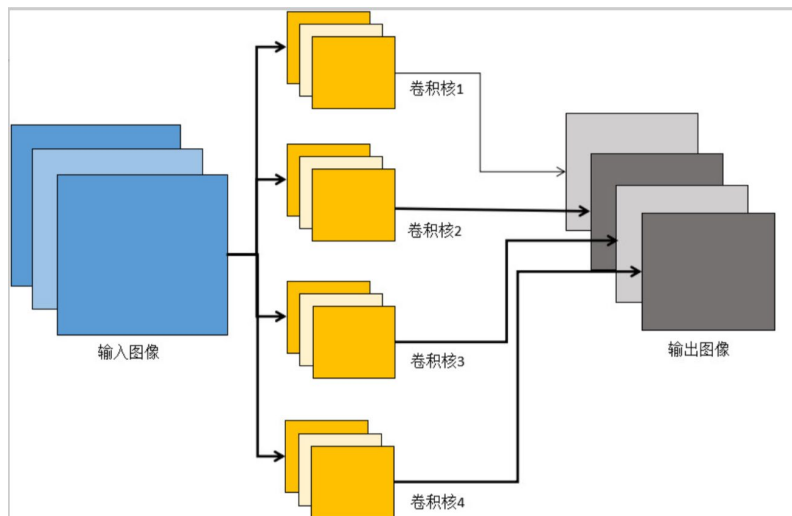
executed in 11ms, finished 15:45:14 2020-06-17

```
tensor([ [ 56.,  72.],
        [104., 120.]])
```

**卷积核通道数 = 输入通道数**



## 多输出通道



输入维度:  $(3, H, W)$

卷积核维度:  $(4, 3, k_h, k_w)$  -- 表示有4个卷积核, 每个卷积核有3个通道

输出维度:  $(4, H_{out}, W_{out})$

## 实现多输出通道

```
1 def corr2d_multi_in_out(X, K):  
2     # 对k的第0维遍历, 每次同输入x做互相关计算。  
3     # 所有结果使用stack函数合并在一起  
4     return torch.stack([corr2d_multi_in(X, k) for k in K])  
5  
6 #X shape: (C_in, H, W)  
7 X = torch.arange(192, dtype=torch.float).view((3, 8, 8))  
8 #K shape: (C_out, C_in, k_h, k_w)  
9 K = torch.arange(108, dtype=torch.float).view((4, 3, 3, 3))  
10 print("kernel shape:", K.shape)  
11  
12 Y = corr2d_multi_in_out(X, K)  
13 print("Y shape:", Y.shape)
```

executed in 30ms, finished 17:09:43 2020-06-17

kernel shape: torch.Size([4, 3, 3, 3])

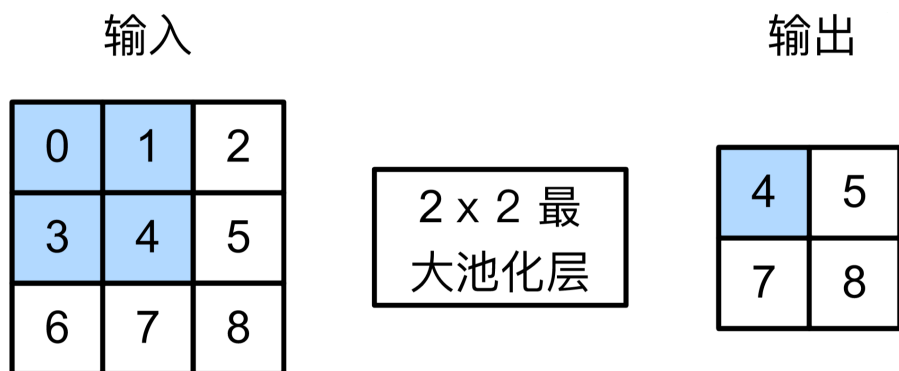
Y shape: torch.Size([4, 6, 6])

**输出通道数 = 卷积核个数**



## 最大池化、平均池化

- 减少输出的高和宽
- 缓解卷积层对位置的过度敏感性



## 自定义池化

```
1 def pool2d(X, pool_size, mode='max'):
2     X = X.float()
3     p_h, p_w = pool_size
4     Y = torch.zeros(X.shape[0] - p_h + 1, X.shape[1] - p_w + 1)
5     for i in range(Y.shape[0]):
6         for j in range(Y.shape[1]):
7             if mode == 'max':
8                 Y[i, j] = X[i: i + p_h, j: j + p_w].max()
9             elif mode == 'avg':
10                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
11     return Y
```

executed in 10ms, finished 17:24:32 2020-06-17

```
1 X = torch.tensor([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
2
3 print("最大池化")
4 print(pool2d(X, (2, 2)))
5
6 print("平均池化")
7 print(pool2d(X, (2, 2), 'avg'))
```

executed in 9ms, finished 17:24:32 2020-06-17

最大池化

```
tensor([[4., 5.],
        [7., 8.]])
```

平均池化

```
tensor([[2., 3.],
        [5., 6.]])
```



# CIFAR-10数据集

包含60,000张 $32 \times 32$ 的彩色图像，维度  $(3, 32, 32)$

一共有10类，每类有6,000个图像=5,000个训练集+1,000个测试集

airplane



automobile



bird



cat



deer



dog



frog



horse



ship



truck

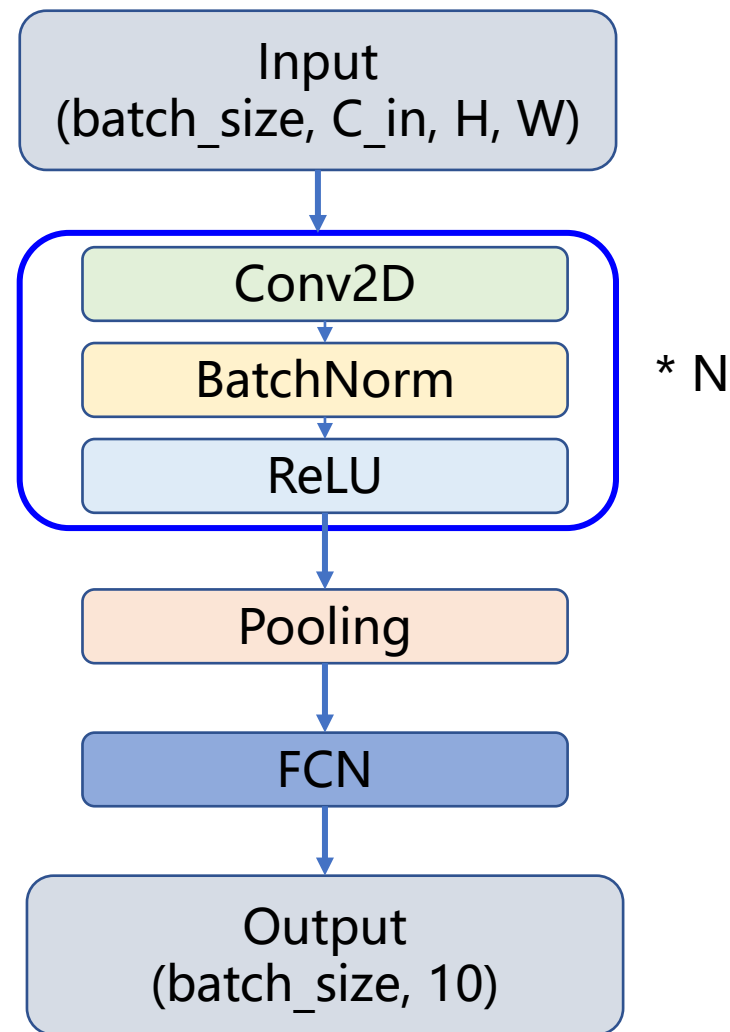




# 模型介绍

## ■ 用以下2种方式实现卷积神经网络模型，实现CIFAR-10图像分类任务

- 自定义的卷积层 (N=1)
- PyTorch已封装的卷积层 (N=3)





# 自定义卷积层

```
1 def corr2d(X,K):
2     """
3     X: 输入, shape (batch_size,H,W)
4     K: 卷积核, shape (k_h,k_w)
5     单通道
6     """
7     batch_size,H, W = X.shape
8     k_h, k_w = K.shape
9     #初始化结果矩阵
10    Y = torch.zeros((batch_size,H - k_h + 1,W - k_w + 1)).to(device)
11    for i in range(Y.shape[1]):
12        for j in range(Y.shape[2]):
13            Y[:,i, j] = (X[:,i:i + k_h, j:j + k_w] * K).sum()
14    return Y
15
16 def corr2d_multi_in(X, K):
17     #输入X: 维度(batch_size,C_in,H,W)
18     #卷积核K: 维度(C_in,k_h,k_w)
19     #输出: 维度(batch_size,H_out,W_out)
20     res = corr2d(X[:,0, :, :], K[0, :, :])
21     for i in range(1, X.shape[1]):
22         #按通道相加
23         res += corr2d(X[:,i, :, :], K[i, :, :])
24     return res
25
26 def corr2d_multi_in_out(X, K):
27     # X: shape (batch_size,C_in,H,W)
28     # K: shape (C_out,C_in,h,w)
29     #Y: shape(batch_size,C_out,H_out,W_out)
30     return torch.stack([corr2d_multi_in(X, k) for k in K],dim=1)
```

```
1 class MyConv2D(nn.Module):
2     def __init__(self, in_channels, out_channels, kernel_size):
3         super(MyConv2D, self).__init__()
4         #初始化卷积层的2个参数: 卷积核、偏差
5         if isinstance(kernel_size,int):
6             kernel_size = (kernel_size,kernel_size)
7         self.weight = nn.Parameter(torch.randn((out_channels,in_channels)+kernel_size))
8         self.bias = nn.Parameter(torch.randn(out_channels,1,1))
9
10    def forward(self, x):
11        """
12        x: 输入图片, 维度(batch_size,C_in,H,W)
13        """
14        return corr2d_multi_in_out(x, self.weight) + self.bias
```

自定义多通道卷积操作

封装

自定义卷积层MyConv2D



```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')
```

## 参数

|    | 名称           | 含义              | 类型        |
|----|--------------|-----------------|-----------|
| 必填 | in_channels  | 输入通道数           | int       |
|    | out_channels | 输出通道数           | int       |
|    | kernel_size  | 卷积核大小           | int或tuple |
| 可选 | stride       | 步长, 默认1         | int或tuple |
|    | padding      | 填充, 默认0         | int或tuple |
|    | dilation     | 空洞率, 默认1        | int或tuple |
|    | group        | 输入和输出通道分组数, 默认1 | int       |
|    | bias         | 是否添加偏值, 默认True  | bool      |
|    | padding_mode | 填充模式, 默认zero    | str       |

## 输入和输出

input: 维度 ( $batch\_size, C_{in}, H_{in}, W_{in}$ )

output: 维度 ( $batch\_size, C_{out}, H_{out}, W_{out}$ )

- 例如: 输入为CIFAR数据, batch\_size=64 则输入维度为 (64, 3, 32, 32)



# 2种方案实现

## 自定义卷积层

```
1 class MyConvModule(nn.Module):
2     def __init__(self):
3         super(MyConvModule, self).__init__()
4         # 定义三层卷积
5         self.conv = nn.Sequential(
6             MyConv2D(in_channels=3, out_channels=32, kernel_size=3),
7             nn.BatchNorm2d(32),
8             nn.ReLU(inplace=True)
9         )
10        # 输出层, 将通道数变为分类数量
11        self.fc = nn.Linear(32, num_classes)
12
13    def forward(self, X):
14        # 图片先经过三层卷积, 输出维度(batch_size, C_out, H, W)
15        out = self.conv(X)
16        # 使用平均池化层将图片的大小变为1x1
17        out = F.avg_pool2d(out, 30)
18        # 将张量out从shape batch x 32 x 1 x 1 变为 batch x 32
19        out = out.squeeze()
20        # 输入到全连接层将输出的维度变为10
21        out = self.fc(out)
22        return out
```

## PyTorch封装卷积层

```
1 class ConvModule(nn.Module):
2     def __init__(self):
3         super(ConvModule, self).__init__()
4         # 定义一个三层卷积
5         self.conv = nn.Sequential(
6             nn.Conv2d(in_channels=3, out_channels=32,
7                       kernel_size=3, stride=1, padding=0),
8             nn.BatchNorm2d(32),
9             nn.ReLU(inplace=True),
10            nn.Conv2d(in_channels=32, out_channels=64,
11                      kernel_size=3, stride=1, padding=0),
12            nn.BatchNorm2d(64),
13            nn.ReLU(inplace=True),
14            nn.Conv2d(in_channels=64, out_channels=128,
15                      kernel_size=3, stride=1, padding=0),
16            nn.BatchNorm2d(128),
17            nn.ReLU(inplace=True)
18        )
19        # 输出层, 将通道数变为分类数量
20        self.fc = nn.Linear(128, num_classes)
21
22    def forward(self, X):
23        # 图片先经过三层卷积, 输出维度(batch_size, C_out, H, W)
24        out = self.conv(X)
25        # 使用平均池化层将图片的大小变为1x1
26        out = F.avg_pool2d(out, 26)
27        # 将张量out从shape batch x 128 x 1 x 1 变为 batch x 128
28        out = out.squeeze()
29        # 输入到全连接层将输出的维度变为10
30        out = self.fc(out)
31        return out
```



# 定义训练和测试函数

## 训练函数

```
1 def train_epoch(net, data_loader, device):
2
3     net.train() # 指定当前为训练模式
4     train_batch_num = len(data_loader) # 记录共有多少个batch
5     total_loss = 0 # 记录Loss
6     correct = 0 # 记录共有多少个样本被正确分类
7     sample_num = 0 # 记录样本总数
8
9     # 遍历每个batch进行训练
10    for batch_idx, (data, target) in enumerate(data_loader):
11        # 将图片放入指定的device中
12        data = data.to(device).float()
13        # 将图片标签放入指定的device中
14        target = target.to(device).long()
15        # 将当前梯度清零
16        optimizer.zero_grad()
17        # 使用模型计算出结果
18        output = net(data)
19        # 计算损失
20        loss = criterion(output, target)
21        # 进行反向传播
22        loss.backward()
23        optimizer.step()
24        # 累加Loss
25        total_loss += loss.item()
26        # 找出每个样本值最大的idx, 即代表预测此图片属于哪个类别
27        prediction = torch.argmax(output, 1)
28        # 统计预测正确的类别数量
29        correct += (prediction == target).sum().item()
30        # 累加当前的样本总数
31        sample_num += len(prediction)
32    # 计算平均的Loss与准确率
33    loss = total_loss / train_batch_num
34    acc = correct / sample_num
35    return loss, acc
```

模型训练  
反向传播更新参数

计算Loss和Acc

## 测试函数

```
1 def test_epoch(net, data_loader, device):
2     net.eval() # 指定当前模式为测试模式
3     test_batch_num = len(data_loader)
4     total_loss = 0
5     correct = 0
6     sample_num = 0
7     # 指定不进行梯度变化
8     with torch.no_grad():
9         for batch_idx, (data, target) in enumerate(data_loader):
10             data = data.to(device).float()
11             target = target.to(device).long()
12             output = net(data)
13             loss = criterion(output, target)
14             total_loss += loss.item()
15             prediction = torch.argmax(output, 1)
16             correct += (prediction == target).sum().item()
17             sample_num += len(prediction)
18     loss = total_loss / test_batch_num
19     acc = correct / sample_num
20     return loss, acc
```

在测试集上验证，计算Loss和Acc



# 读取数据

```
1 data_dir = './data' # 指定数据的位置
2 # 定义一个transform操作, 用户将torch中的数据转换为可以输入到我们模型的形式
3 transform = transforms.Compose(
4     [transforms.ToTensor(), # 首先将数据转换为Tensor
5      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]) # 将数据进行归一化, 前一个参数代表mean, 后一个
6 # 获取cifar-10数据集并进行transform
7 cifar_train = torchvision.datasets.CIFAR10(root=data_dir, train=True, download=True, transform=transform)
8 cifar_test = torchvision.datasets.CIFAR10(root=data_dir, train=False, download=True, transform=transform)
9 # print(len(cifar_train))
10 # cifar-10数据集对应的10个类别
11 classes = ('plane', 'car', 'bird', 'cat',
12            'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
13 num_classes = 10 # 共十类
14 epochs = 100 # 训练多少轮
15 lr = 0.001 # 学习率
16 batch_size = 512 # batch大小
17 device = torch.device("cuda:1") # 指定device为0号GPU, 若使用cpu则填写 "cpu"
18 # 生成dataLoader
19 cifar_trainloader = torch.utils.data.DataLoader(cifar_train, batch_size=batch_size,
20                                                  shuffle=True, num_workers=0)
21 cifar_testloader = torch.utils.data.DataLoader(cifar_test, batch_size=512,
22                                                  shuffle=True, num_workers=0)
23 print(len(cifar_trainloader))
24 # 初始化模型
25
26 # net = MyConvModule().to(device)
27 net = ConvModule().to(device)
28
29 # 使用多元交叉熵损失
30 criterion = nn.CrossEntropyLoss()
31 # 使用Adam优化器
32 optimizer = optim.Adam(net.parameters(), lr=lr)
```

使用2种方式定义模型

## 读取CIFAR-10数据

## 设置模型参数

- 学习率
- Batch\_size
- 训练轮数
- 优化器
- 损失函数



# 自定义卷积实验结果

## 模型训练并验证

```
1 # 存储每一个epoch的Loss与acc的变化, 便于后面可视化
2 train_loss_list = []
3 train_acc_list = []
4 test_loss_list = []
5 test_acc_list = []
6
7 # 进行训练
8 for epoch in range(epochs):
9     # 在训练集上训练
10    train_loss, train_acc = train_epoch(net, data_loader=cifar_trainloader, device=device)
11    # 在测试集上验证
12    test_loss, test_acc = test_epoch(net, data_loader=cifar_testloader, device=device)
13    # 保存各个指标
14    train_loss_list.append(train_loss)
15    train_acc_list.append(train_acc)
16    test_loss_list.append(test_loss)
17    test_acc_list.append(test_acc)
18    print(f"epoch:{epoch}\t train_loss:{train_loss:.4f} \t"
19          f"train_acc:{train_acc} \t"
20          f"test_loss:{test_loss:.4f} \t test_acc:{test_acc}")
```

execution queued 22:21:21 2020-06-18

```
epoch-batch:0-0  train_loss:2.3689      train_acc:0.0000
epoch-batch:0-1  train_loss:2.3378      train_acc:0.1000
epoch-batch:0-2  train_loss:2.3240      train_acc:0.1333
epoch-batch:0-3  train_loss:2.2734      train_acc:0.2500
epoch-batch:0-4  train_loss:2.2889      train_acc:0.2400
```

## 耗时原因分析

```
def corr2d(X,K):
    """
    X: 输入, shape (batch_size,H,W)
    K: 卷积核, shape (k_h,k_w)
    单通道
    """
    batch_size,H, W = X.shape
    k_h, k_w = K.shape
    #初始化结果矩阵
    Y = torch.zeros((batch_size,H - k_h + 1,W - k_w + 1)).to(device)
    for i in range(Y.shape[1]):
        for j in range(Y.shape[2]):
            Y[:,i, j] = (X[:,i: i + k_h, j: j + k_w] * K).sum()
    return Y
```

- 耗时太长, 1轮耗时>5h
- 分析原因: 卷积操作使用for循环实现, 而不是矩阵操作



# PyTorch封装卷积实验结果

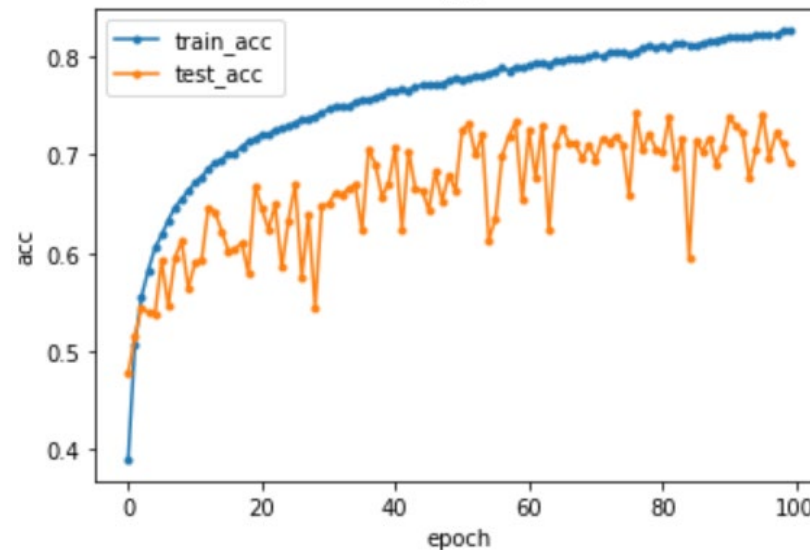
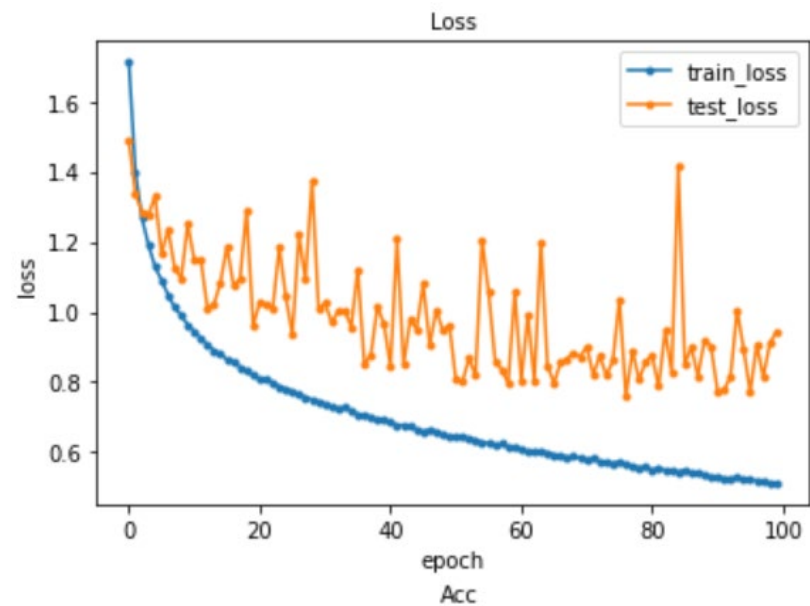
## 模型训练并验证

100轮耗时: 1h

```
1 # 存储每一个epoch的loss与acc的变化, 便于后面可视化
2 train_loss_list = []
3 train_acc_list = []
4 test_loss_list = []
5 test_acc_list = []
6
7 # 进行训练
8 for epoch in range(epochs):
9     # 在训练集上训练
10    train_loss, train_acc = train_epoch(net, data_loader=cifar_trainloader, device=device)
11    # 在测试集上验证
12    test_loss, test_acc = test_epoch(net, data_loader=cifar_testloader, device=device)
13    # 保存各个指标
14    train_loss_list.append(train_loss)
15    train_acc_list.append(train_acc)
16    test_loss_list.append(test_loss)
17    test_acc_list.append(test_acc)
18    print(f"epoch:{epoch}\t train_loss:{train_loss:.4f} \t"
19          f"train_acc:{train_acc} \t"
20          f"test_loss:{test_loss:.4f} \t test_acc:{test_acc}")
```

executed in 1h 7m 5s, finished 16:30:17 2020-06-18

|         |                   |                   |                  |                 |
|---------|-------------------|-------------------|------------------|-----------------|
| epoch:0 | train_loss:1.7119 | train_acc:0.38988 | test_loss:1.4879 | test_acc:0.4774 |
| epoch:1 | train_loss:1.3972 | train_acc:0.50722 | test_loss:1.3381 | test_acc:0.5147 |
| epoch:2 | train_loss:1.2678 | train_acc:0.55578 | test_loss:1.2836 | test_acc:0.5444 |
| epoch:3 | train_loss:1.1904 | train_acc:0.58214 | test_loss:1.2779 | test_acc:0.5396 |
| epoch:4 | train_loss:1.1289 | train_acc:0.60496 | test_loss:1.3294 | test_acc:0.5369 |
| epoch:5 | train_loss:1.0899 | train_acc:0.6181  | test_loss:1.1660 | test_acc:0.592  |





## 1. 卷积神经网络

- 卷积基本操作
- 手动实现卷积层
- 图像分类示例

## 2. 空洞卷积

- 空洞卷积的概念
- 优点与适用性
- 缺点与解决方法
- 使用PyTorch实现与实验结果

## 3. 残差网络

- 解决的问题
- 网络结构
- 使用PyTorch实现与实验结果

## 4. 实验要求

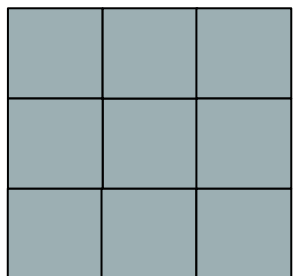
- 数据集介绍
- 卷积实验
- 空洞卷积实验
- 残差网络实验



# 空洞卷积(dilated convolution)

普通卷积

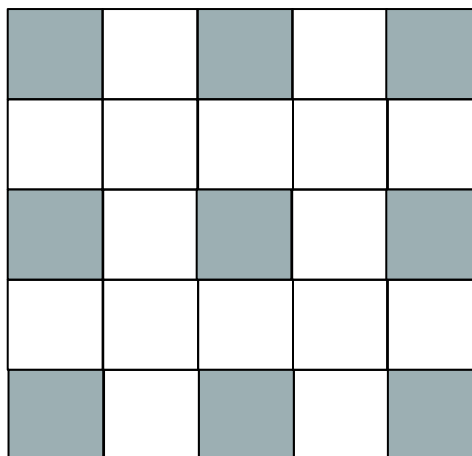
3 x 3



VS

空洞卷积

3 x 3, dilation=2

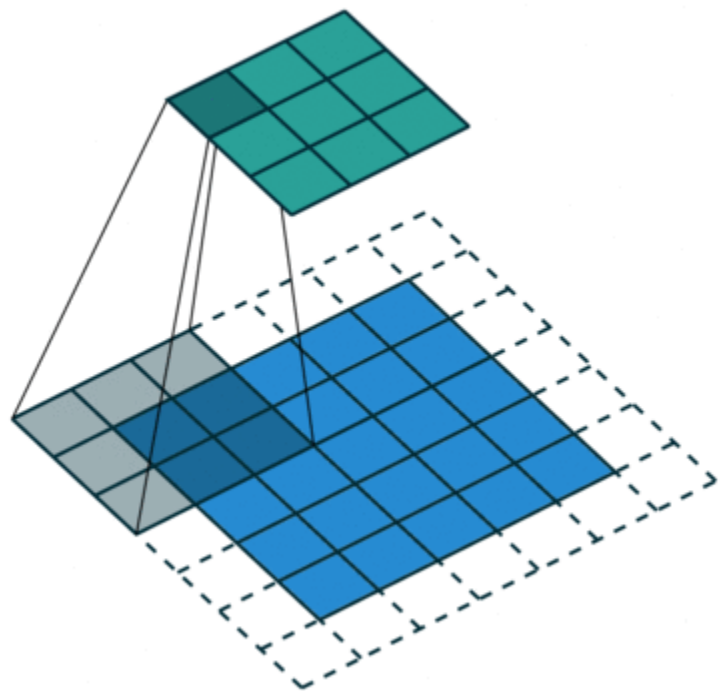


- 灰色部分为卷积核权重，与左侧相同；白色部分为空，值为0
- dilation rate: 空洞率，空洞卷积权重值的间隔为dilation rate - 1
- 当空洞率为1时，退化为普通卷积

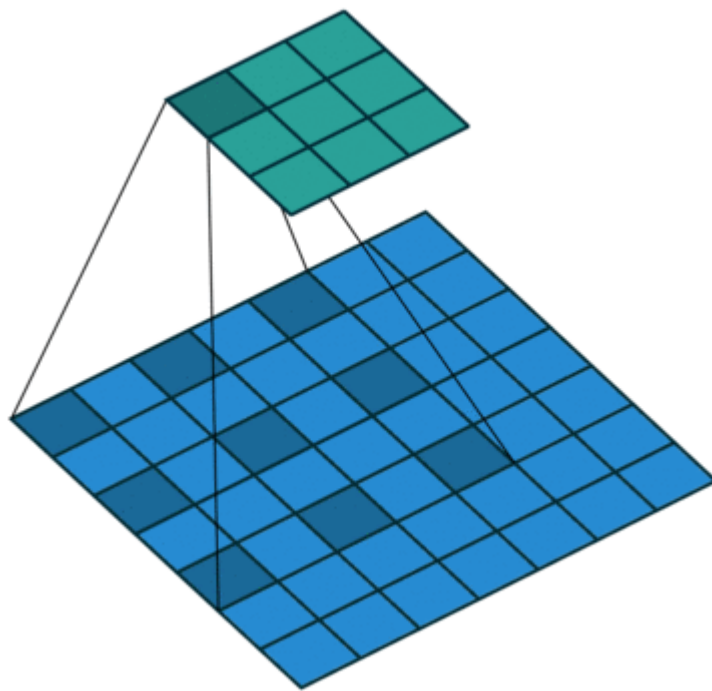


# 空洞卷积(dilated convolution)

普通卷积



空洞卷积



- 空洞后的卷积核的空间(感受野)大小:

$$k = (n - 1) \times d + 1$$

$n$  = 原卷积核大小

$d$  = dilation rate

- 卷积后图片大小计算公式:

$$O = \lfloor \frac{(I - k + 2P)}{S} \rfloor + 1$$

$I$  = 输入图片大小

$P$  = 填充大小

$S$  = 步长

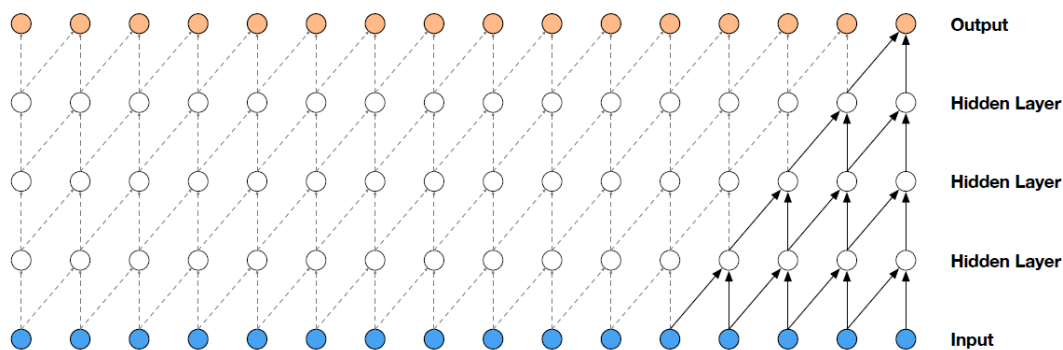
图片出处: [https://github.com/vdumoulin/conv\\_arithmetic](https://github.com/vdumoulin/conv_arithmetic)



# 空洞卷积(dilated convolution)

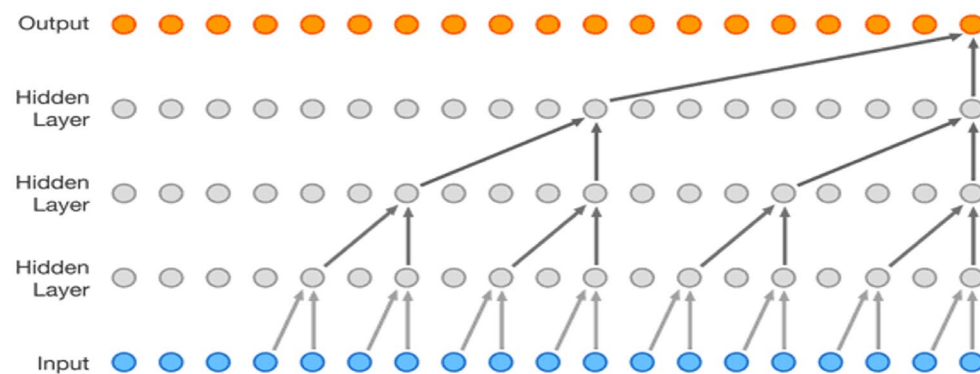
## 优点与适用性

- 在不增加参数的情况下**增大感受野**，适用于图片size较大，或需要快速感受全局信息的情况



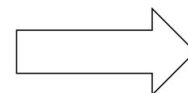
多层普通卷积

卷积核为大小2，四层卷积，普通卷积可以覆盖**5**个像素点，而使用空洞卷积可以覆盖**16**个像素点



多层空洞卷积

- 通过设置不同的dilation rate捕获**多尺度**上下文信息，适用于需要捕获图片的多层次的情况，如语义分割等。

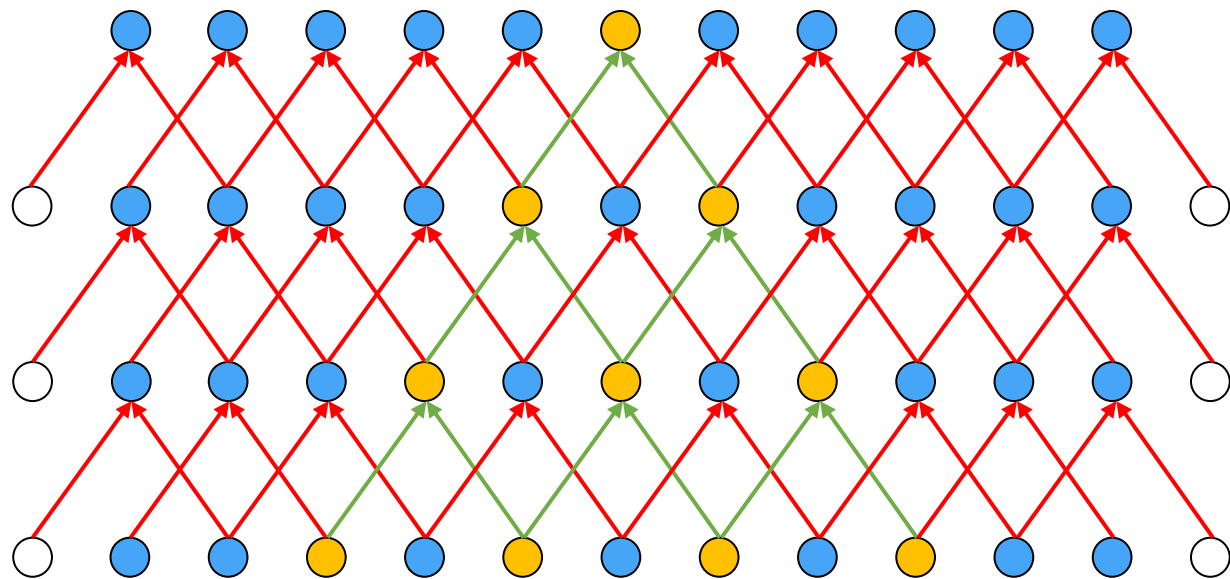




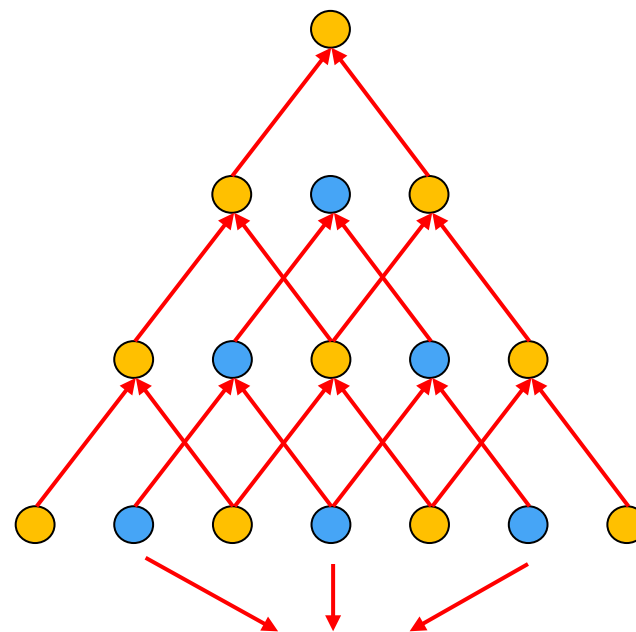
# 空洞卷积(dilated convolution)

## 存在的问题

空洞卷积存在网格效应



三层空洞卷积  
dilation=2



堆叠后漏掉了三个点的信息

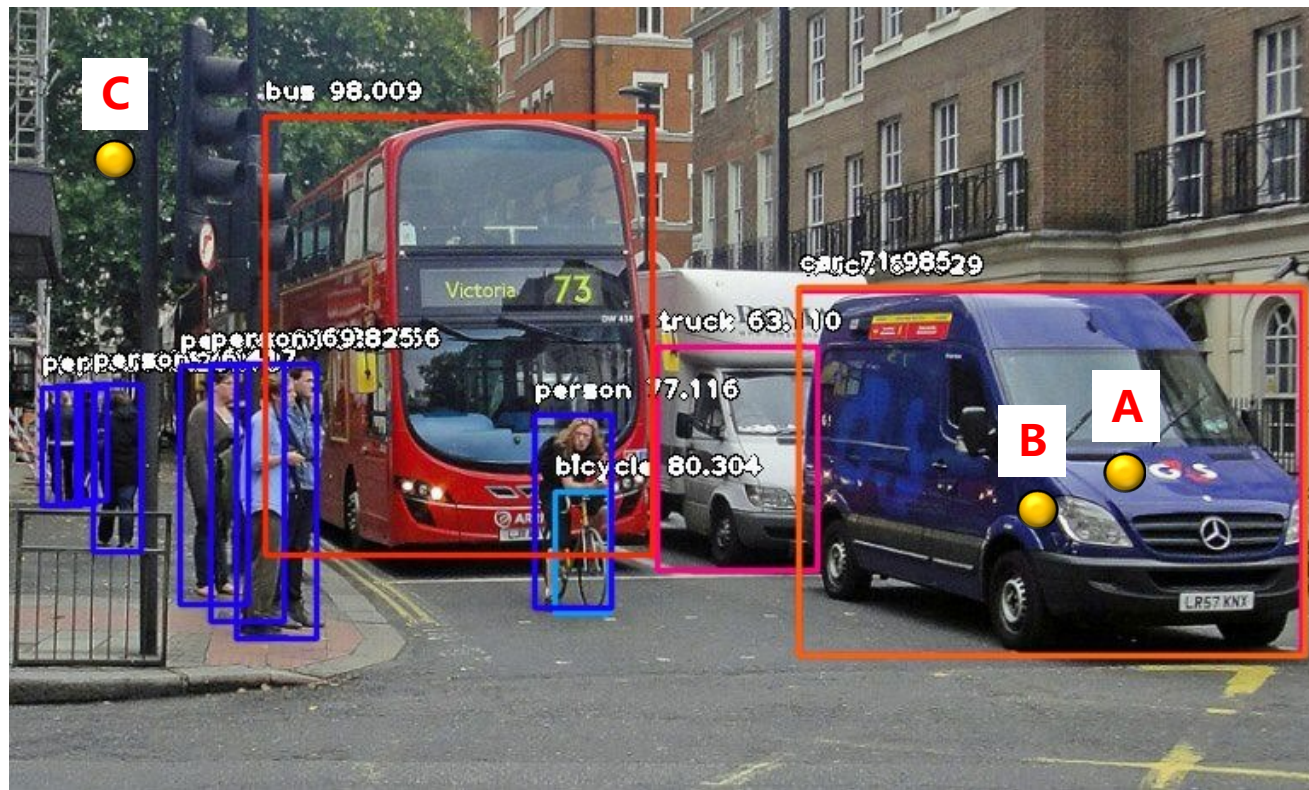


# 空洞卷积(dilated convolution)

## 存在的问题

远距离的点之间的信息可能不相关

空洞卷积虽然是被设计用来捕获更**远距**离的信息的，但是这样一味的使用大的dilation就会**忽略掉近距离的信息**，如何能**同时处理远距离信息和临近区域的信息**是设计好空洞卷积的关键之一。



点B明显与**近距离的点A更相关**，与左上角的点C关系不大，如果dilation过大则容易**忽略点A**的信息而降低模型效果



# 空洞卷积(dilated convolution)

## 解决方法

### Hybrid Dilated Convolution (HDC)

一、叠加卷积的 **dilation rate** 不能有大于1的公约数。比如 [2, 4, 6] 则不是一个好的三层卷积，依然会出现 gridding effect。 **解决网格效应**

二、我们将 dilation rate 设计成**锯齿状结构**，例如 [1, 2, 5, 1, 2, 5] 循环结构。

**同时捕获远近信息**

三、满足

$$M_i = \max[M_{i+1} - 2r_i, M_{i+1} - 2(M_{i+1} - r_i), r_i]$$

$$M_2 \leq k$$

$$M_i = \max[M_{i+1} - 2r_i, M_{i+1} - 2(M_{i+1} - r_i), r_i]$$

目标:  $M_2 \leq k$

$r_i$  指第i层的dilation rate

$M_i$ 指的是第i层的dilation rate的最大值

默认最后一层的 $M_n = r_n$

$k$  = kernel size

dilation rate [1, 2, 5] with 3 x 3 kernel

$$M_3 = r_3 = 5$$

$$\begin{aligned} M_2 &= \max\{M_3 - 2r_2, M_3 - 2(M_3 - r_2), r_2\} \\ &= \max\{1, -1, 2\} = 2 \leq k = 3 \end{aligned}$$

满足条件



# 空洞卷积(dilated convolution)

## 使用PyTorch实现空洞卷积

Conv2d

直接使用torch.nn.Conv2d, 选择不同的dilation即可

```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0,  
dilation=1, groups=1, bias=True, padding_mode='zeros')
```

[SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size  $(N, C_{\text{in}}, H, W)$  and output  $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$  can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where  $\star$  is the valid 2D **cross-correlation** operator,  $N$  is a batch size,  $C$  denotes a number of channels,  $H$  is a height of input planes in pixels, and  $W$  is width in pixels.



# 空洞卷积(dilated convolution)

## 空洞卷积的实现

实现dilation为1,2,5的三层空洞卷积

```
class DilatedConvModule(nn.Module):
    def __init__(self):
        super(DilatedConvModule, self).__init__()
        # 定义一个空洞率为1,2,5的三层空洞卷积
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=32,
                      kernel_size=3, stride=1,
                      padding=0, dilation=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=32, out_channels=64,
                      kernel_size=3, stride=1,
                      padding=0, dilation=2),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=64, out_channels=128,
                      kernel_size=3, stride=1,
                      padding=0, dilation=5),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True)
        )
        # 输出层, 将通道数变为分类数量
        self.fc = nn.Linear(128, num_classes)
```

```
def forward(self, X):
    # 图片先经过三层空洞卷积
    out = self.conv(X)
    # 使用平均池化层将图片的大小变为1x1
    out = F.avg_pool2d(out, 16)
    # 将张量out从shape batch x 128 x 1 x 1 变为 batch x 128
    out = out.squeeze()
    # 输入到全连接层将输出的维度变为10
    out = self.fc(out)
    return out
```

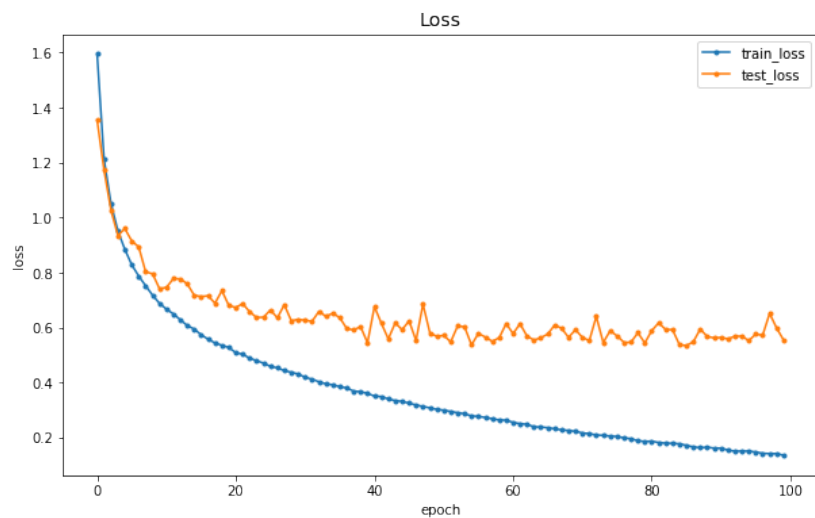
- 图片大小的变化: 32→30→26→16→1
- 池化层为了将图片大小变为1
- 输出层将维度变为类别数, 后续经过处理变为每个类别的概率

只需改变dilation参数, 其他与Conv2d相同



# 空洞卷积(dilated convolution)

## 空洞卷积实验结果

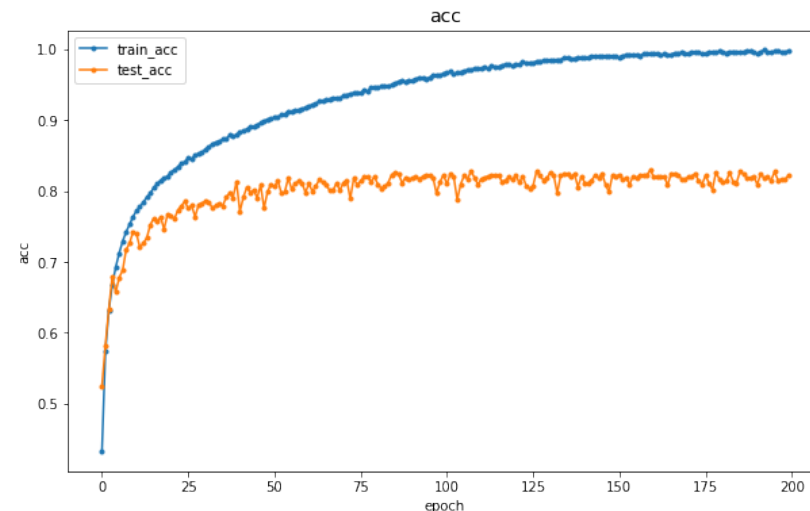


Loss的变化

batchsize=512

lr=0.001

epochs=100



平均准确率的变化

- 模型在训练集上可以达到一个较低的loss和较高的准确率
- 在大约40轮后测试集的Loss基本趋向于收敛
- 测试集的平均准确率达到到了82.92%，说明当前模型在cifar-10上进行分类的效果良好



# 目录

## 1. 卷积神经网络

- 卷积基本操作
- 手动实现卷积层
- 图像分类示例

## 2. 空洞卷积

- 空洞卷积的概念
- 优点与适用性
- 缺点与解决方法
- 使用PyTorch实现与实验结果

## 3. 残差网络

- 解决的问题
- 网络结构
- 使用PyTorch实现与实验结果

## 4. 实验要求

- 数据集介绍
- 卷积实验
- 空洞卷积实验
- 残差网络实验



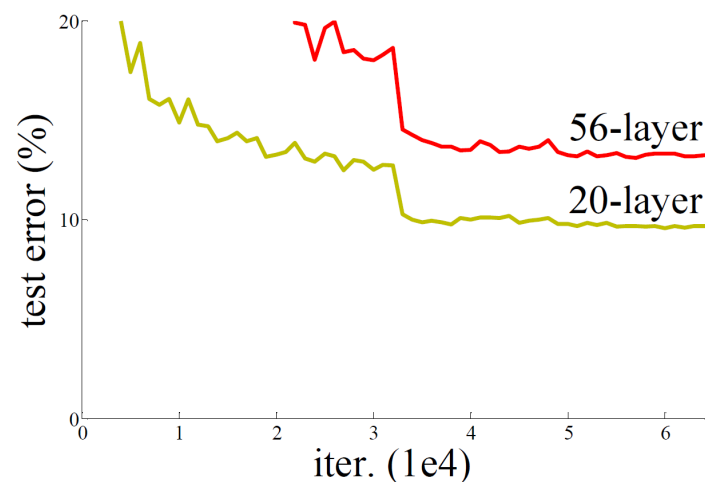
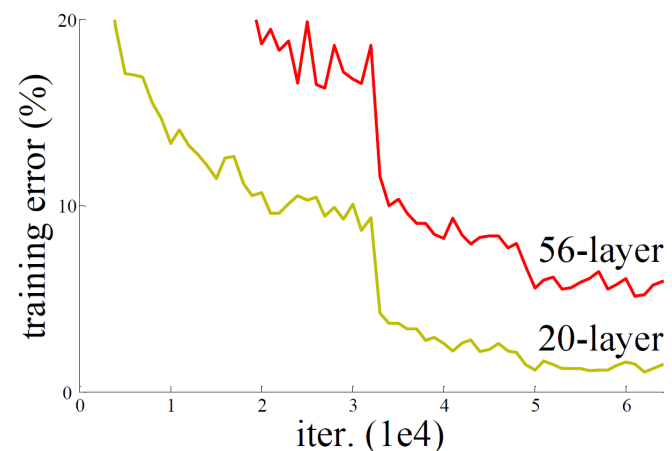
# 残差网络 (Residual Networks, ResNets)

## 堆叠多层卷积

理论上，深层的网络效果不会比浅层网络差，因为深层网络的前几层理论可以学习到一个输入等于输出的映射，然后后几层等于浅层网络，则效果至少相同

右侧的实验结果表明实际上深层的网络效果反而更差，且训练集的效果也很差，说明不是过拟合

反向传播路径太长，训练难度增加



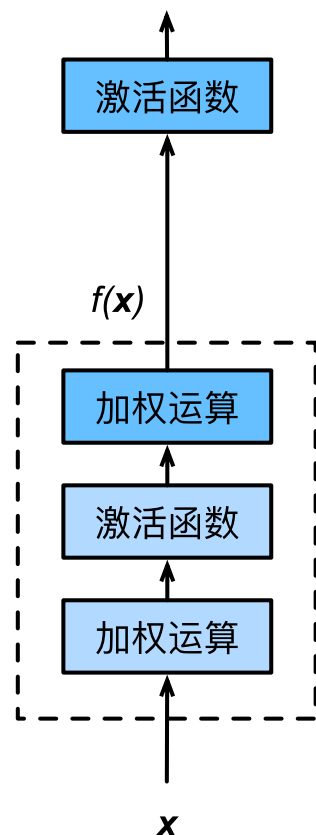
使用20层和56层卷积在CIFAR10数据集上的实验结果



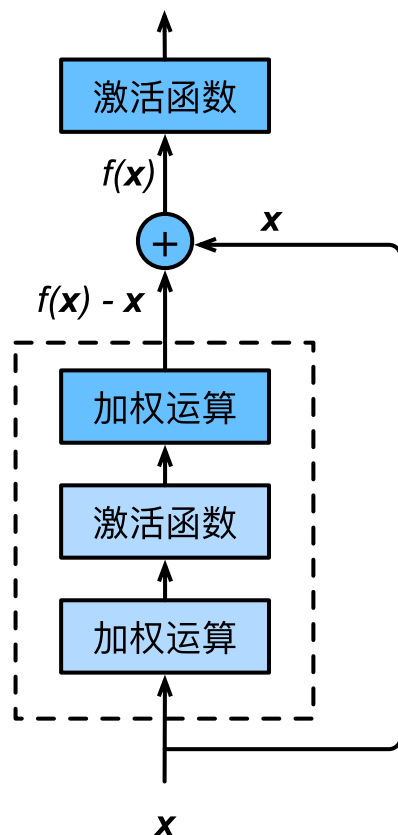
# 残差网络 (Residual Networks, ResNets)

## 残差网络结构

用来解决深层网络训练难度过大的问题



普通网络结构



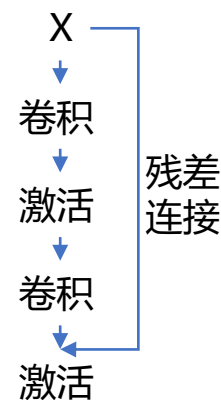
残差网络结构

```
def forward(self, X):  
    Y = F.relu(self.conv1(X))  
    Y = self.conv2(Y)  
    return F.relu(Y)
```

两层卷积代码

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

加入残差结构后





# 残差网络 (Residual Networks, ResNets)

## 残差网络的实现

```
class ResidualBlock(nn.Module):
    def __init__(self, inchannel, outchannel, stride=1):
        super(ResidualBlock, self).init ()
        # 正常卷积部分, 堆叠了两层卷积
        self.left = nn.Sequential(
            nn.Conv2d(inchannel, outchannel,
                      kernel_size=3, stride=stride,
                      padding=1, bias=False),
            nn.BatchNorm2d(outchannel),
            nn.ReLU(inplace=True),
            nn.Conv2d(outchannel, outchannel,
                      kernel_size=3, stride=1,
                      padding=1, bias=False),
            nn.BatchNorm2d(outchannel)
        )
        # 如果上方卷积没有改变size和channel
        # 则不需要对输入进行变化, 故 shortcut为空
        self.shortcut = nn.Sequential()
        # 如果上方卷积改变了size或channel
        # 则使用1x1卷积改变输入的size或channel, 使其保持一致
        if stride != 1 or inchannel != outchannel:
            self.shortcut = nn.Sequential(
                nn.Conv2d(inchannel, outchannel,
                          kernel_size=1, stride=stride,
                          bias=False),
                nn.BatchNorm2d(outchannel)
            )
```

正常的卷积堆叠

解决不能直接相加的问题

实现残差

```
def forward(self, x):
    # 正常使用卷积操作
    out = self.left(x)
    # 将输入/变换shape后的输入与卷积的输出相加
    out += self.shortcut(x)
    # 经过激活函数后输出
    out = F.relu(out)
    return out
```

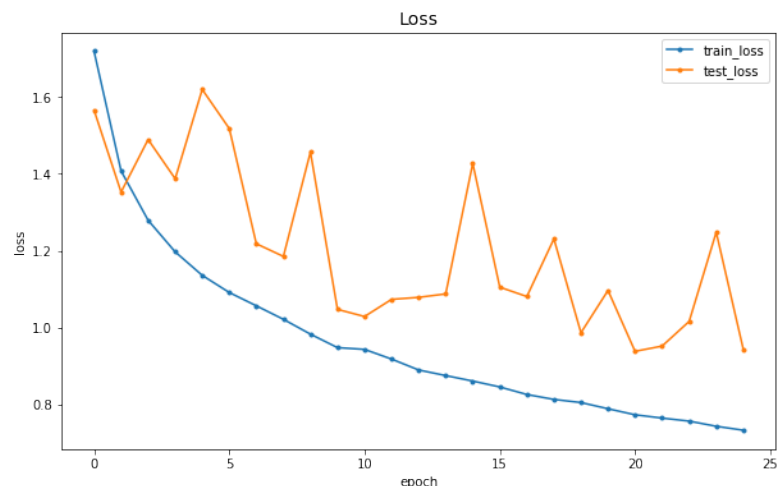
注意先相加再激活

- 若卷积后的结果与输入X的shape不同, 则不能直接相加
- 当步长不为1, 图片大小发生变化 (padding不随步长改变的情况)
- 当通道数与输入通道不同, 图片通道数发生变化
- 使用一个1x1的卷积核来改变X的shape 使得能与卷积后的结果shape相匹配



# 残差网络 (Residual Networks, ResNets)

## 残差模型实验结果

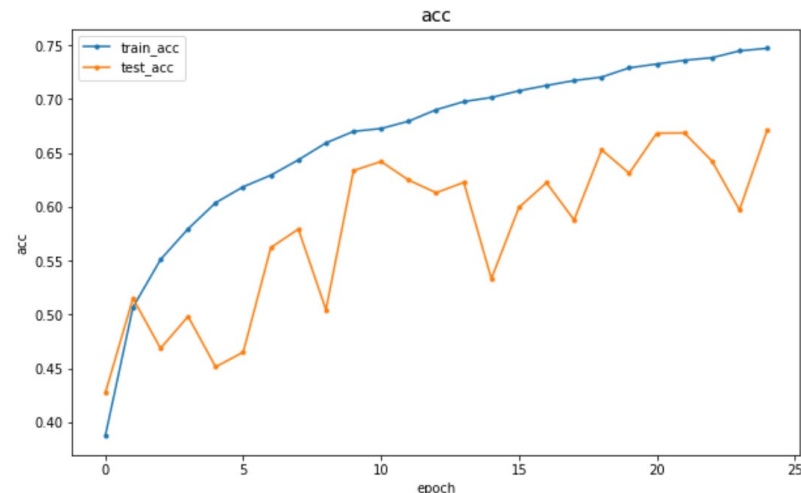


Loss的变化

batchsize=512

lr=0.001

epochs=25



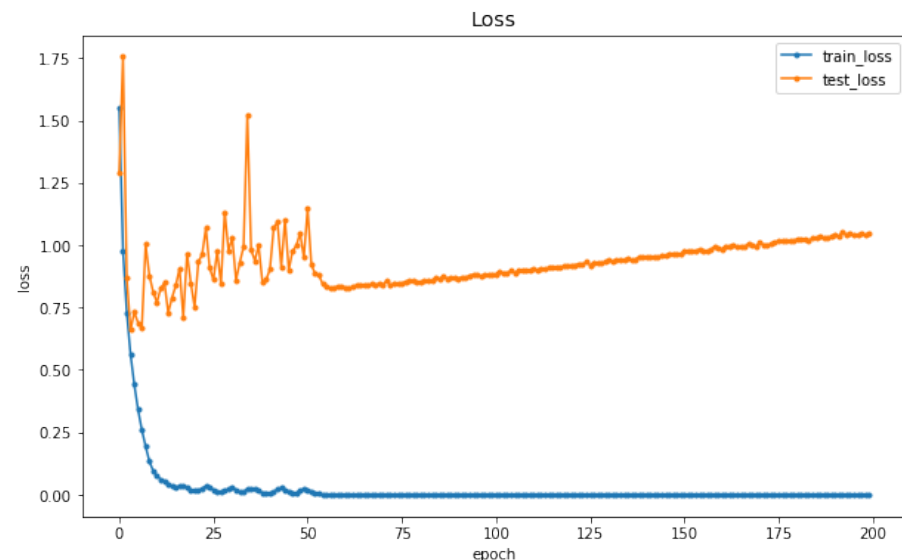
平均准确率的变化

- 在训练集上Loss持续下降，准确率持续上升，说明模型有一定的学习能力
- 在测试集上的效果较差，且波动明显，可能是lr过大，或者模型泛化能力差
- 由于当前卷积只有三层，只是残差操作的一个演示，所以效果并不理想，实际上在浅层网络上一般也不直接使用残差网络



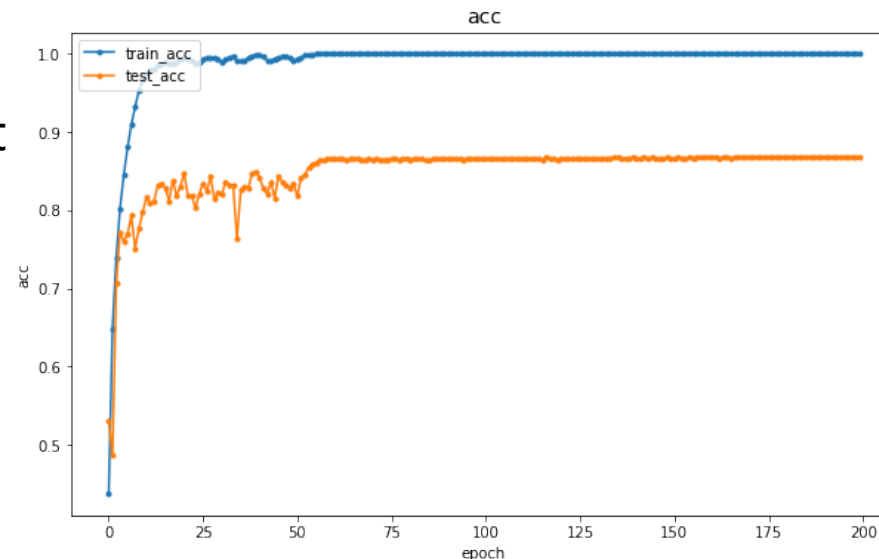
# 残差网络 (Residual Networks, ResNets)

## 残差模型实验结果



Loss的变化

18层Conv的ResNet  
batchsize=512  
lr=0.001  
epochs=200



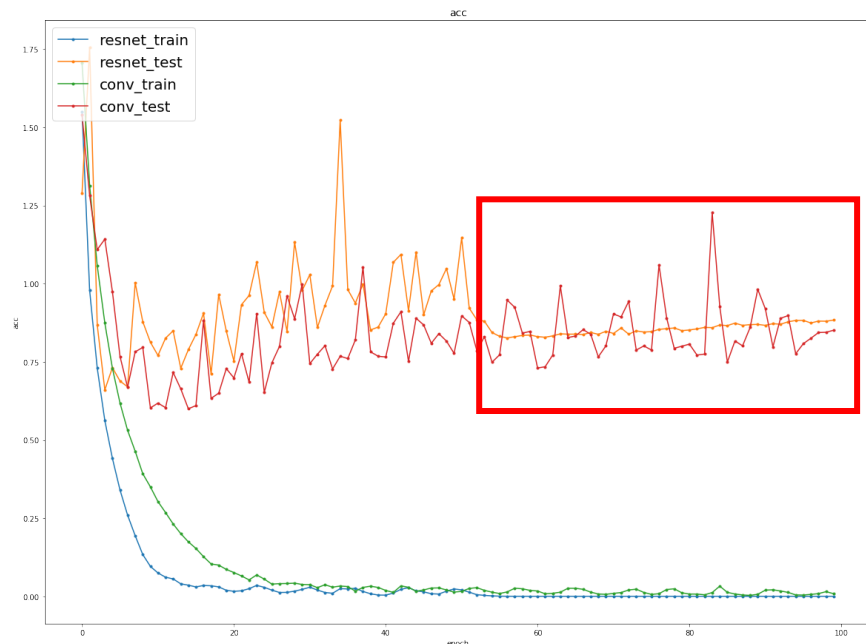
平均准确率的变化

- 模型在训练集上持续训练损失最终降到接近0，准确率也达到了100%，说明模型的容量是足够的
- 在测试集上，Loss在第50轮附近持续上升，而此时训练集的Loss还在持续下降，说明发生了过拟合现象
- 最终在此参数下ResNet-18在测试集上能达到的平均准确率为86.85%



# 残差网络 (Residual Networks, ResNets)

## 残差模型与同等深度卷积的对比



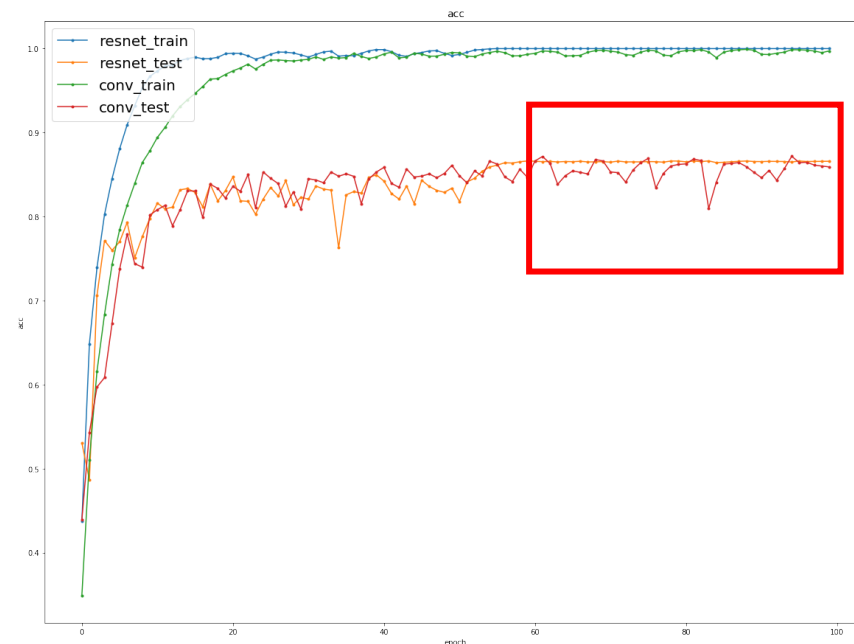
Loss的变化

batchsize=512

lr=0.001

epochs=100

共堆叠18层



平均准确率的变化

- 卷积为红色与绿色曲线，残差为黄色和蓝色曲线
- 可以看到在深层模型下，无论是Loss还是准确率，普通卷积的曲线（红色）比残差的曲线（黄色）在训练后期的波动更大，说明残差网络的加入确实可以帮助模型更好的训练



## 1. 卷积神经网络

- 卷积基本操作
- 手动实现卷积层
- 图像分类示例

## 2. 空洞卷积

- 空洞卷积的概念
- 优点与适用性
- 缺点与解决方法
- 使用PyTorch实现与实验结果

## 3. 残差网络

- 解决的问题
- 网络结构
- 使用PyTorch实现与实验结果

## 4. 实验要求

- 数据集介绍
- 卷积实验
- 空洞卷积实验
- 残差网络实验



# 实验要求

## 数据集——车辆分类数据

- 输入图片，输出对应的类别 **分类问题**
- 共1358张车辆图片
- 分别属于汽车、客车和货车三类
- 汽车：779张
- 客车：218张
- 货车：360张
- 每个类别的后20-30%当作测试集
- 各图片的**大小不一**，需要将图片拉伸到**相同大小**



汽车



客车



货车

## 数据集——照片去雾数据

- 输入带雾图片，输出去雾图片

回归问题

- 共520张图片

- 后20%的图片当作测试集

- 各图片的大小不一，需要将图片拉伸到相同大小

- 实验分析需要有多张照片的原图、模型输出图、数据集中不带雾图的对比



原图（带雾）



输出（去雾）



# 实验要求

## 数据基本处理

# 导入相关的包

```
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
```

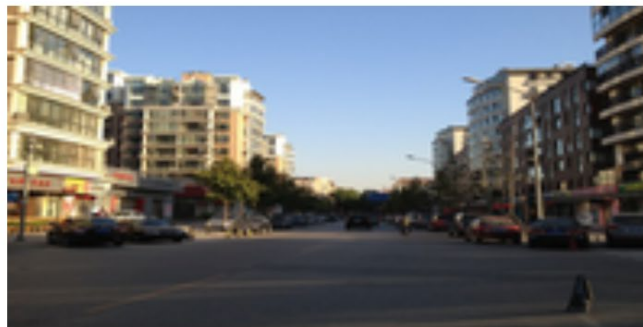
# 读取并展示图片

```
file = Path("D:\MyFile\深度学习数据集\去雾数据集\去雾图片\001.jpg")
img = Image.open(file)
plt.imshow(img)
plt.axis('off')
plt.show()
```



# 对图片进行变形

```
img = img.resize((200,100),Image.ANTIALIAS)
img
```



# 将图片转成numpy矩阵形式

```
img = np.array(img)
print(img.shape)
```

(100, 200, 3)

# 归一化

```
img = img / 255
```



# 实验要求

## 二维卷积实验

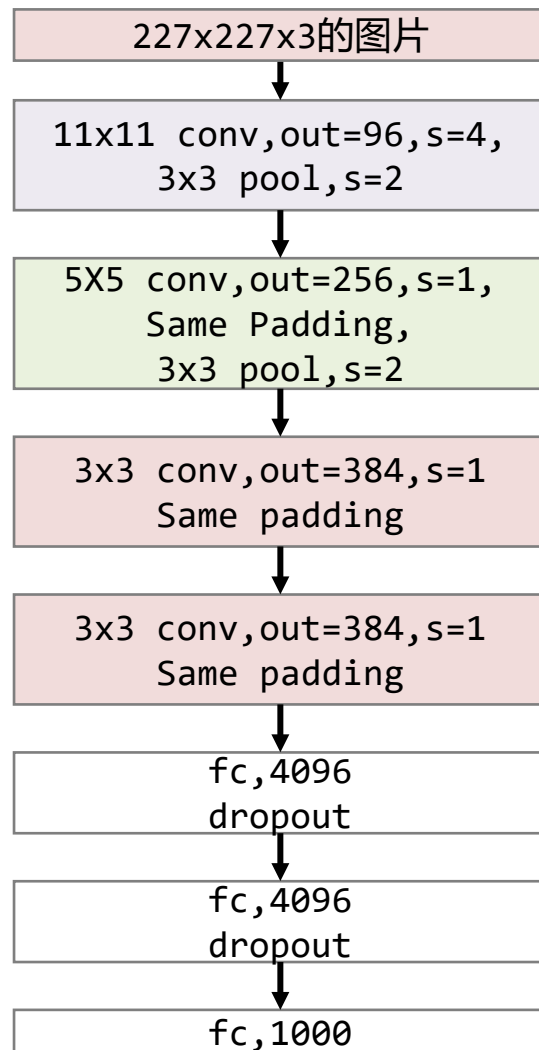
- 手写二维卷积的实现，并在至少一个数据集上进行实验，从训练时间、预测精度、Loss变化等角度分析实验结果（最好使用图表展示）
- 使用torch.nn实现二维卷积，并在至少一个数据集上进行实验，从训练时间、预测精度、Loss变化等角度分析实验结果（最好使用图表展示）
- 不同超参数的对比分析（包括卷积层数、卷积核大小、batchsize、lr等）选其中至少1-2个进行分析
- 使用PyTorch实现经典模型AlexNet并在至少一个数据集进行试验分析（无GPU环境则至少实现模型）
- 使用实验2中的前馈神经网络模型在本次给定数据集上进行实验，并将实验结果与卷积模型结果进行对比分析（选做）



# 实验要求

## AlexNet

激活函数皆为Relu



11x11的卷积和，输出通道为96，步长为4，3x3的最大池化，步长为2

Same Padding: padding后图片大小维持不变

全连接输出维度4096  
加入dropout

输出层输出1000，因为有1000个类别

- 可根据实际数据情况灵活改动各层的步长、卷积核大小、池化等
- 根据最终分类的类别改动输出层的size
- 完整的AlexNet参数约6000万个，若无GPU环境则可以适当将减少参数模型，或者用少量数据能跑通模型即可。
- 关于AlexNet进一步的内容可以参照网上各种文章或者左下角的论文



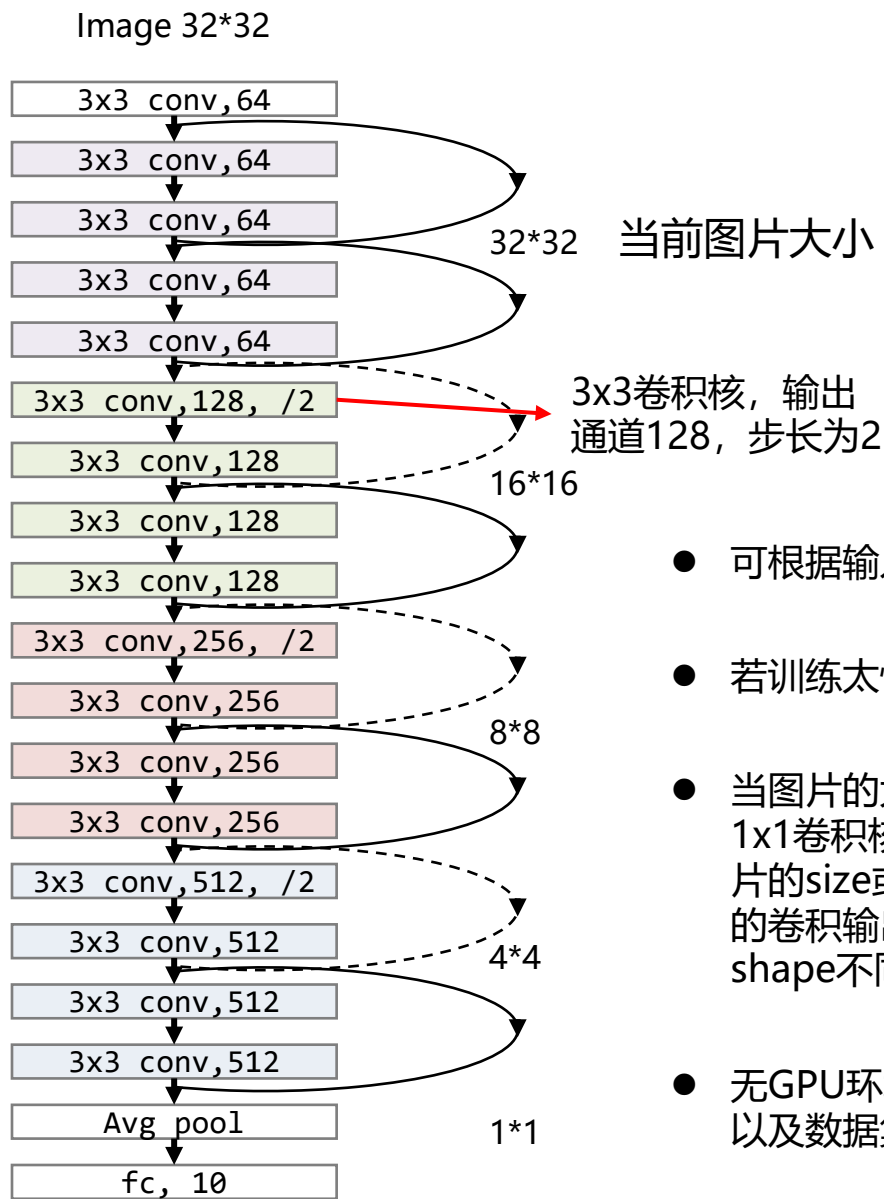
## 空洞卷积实验

- 使用`torch.nn`实现空洞卷积，要求dilation满足HDC条件（如1,2,5）且要堆叠多层并在至少一个数据集上进行实验，从训练时间、预测精度、Loss变化等角度分析实验结果（最好使用图表展示）
- 将空洞卷积模型的实验结果与卷积模型的结果进行分析比对，训练时间、预测精度、Loss变化等角度分析
- 不同超参数的对比分析（包括卷积层数、卷积核大小、不同dilation的选择，batchsize、lr等）选其中至少1-2个进行分析（选做）



## 残差网络实验

- 实现给定结构的残差网络，在**至少一个数据集**上进行实验，从训练时间、预测精度、Loss变化等角度**分析实验结果**（最好使用**图表展示**）
- 将残差网络与空洞卷积相结合，在**至少一个数据集**上进行实验，从训练时间、预测精度、Loss变化等角度**分析实验结果**（最好使用**图表展示**）（选做）



- 可根据输入图片大小灵活调整结构
- 若训练太慢可适当缩减模型层数
- 当图片的大小发生变化时，需要使用1x1卷积核对数据进行卷积来改变图片的size或者通道数，然后再与正常的卷积输出相加，负责两个张量的shape不同无法相加
- 无GPU环境的同学可适当缩减模型以及数据集大小



# 实验提交要求

## ■ 实验报告

- 需要完成之前的任务要求，写清楚实验结果，对实验结果的分析和相关实验参数设置

## ■ 实验报告中涉及的代码

- 完整代码，包括处理数据的代码

## ■ 所有文件放到一个文件夹内压缩

- 名称例如：20120332\_张三\_实验1.zip