卷积神经网络第二部分

1 残差网络

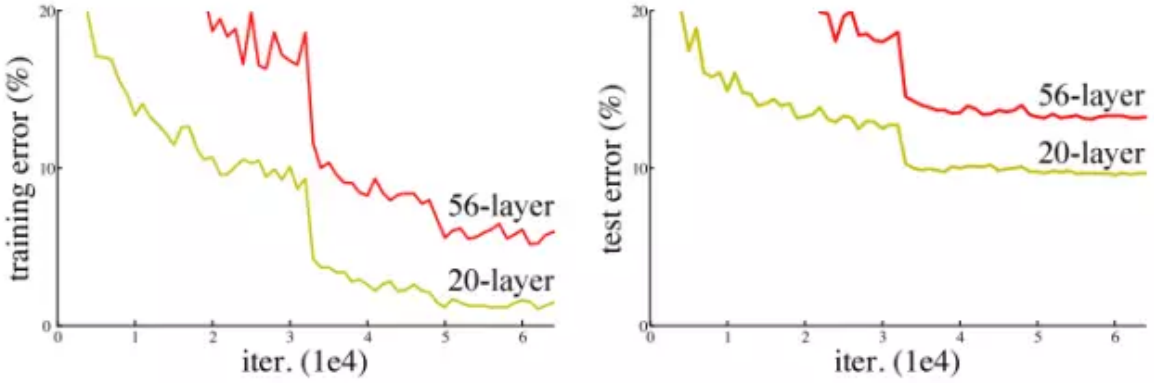
1.1 背景介绍

随着深度学习的不断发展，模型的层数越来越多，网络结构也越来越复杂。但是增加网络的层数之后，训练误差往往不降反升。由此，Kaiming He等人提出了残差网络ResNet来解决上述问题。ResNet是2015年ImageNet比赛的冠军，将识别错误率降低到了3.6%，这个结果甚至超出了正常人眼识别的精度。在ResNet中，提出了一个非常经典的结构—残差块（Residual block）。

自从2015年ResNet进入人们的视线，并引发人们思考之后，许多研究界人员已经开始研究其成功的秘诀，并在架构中纳入了许多新的改进。ResNet可以实现高达数百，甚至数千个层的训练，且仍能获得超赞的性能。

自从AlexNet投入使用以来，最先进的卷积神经网络（CNN）架构越来越深。虽然AlexNet只有5层卷积层，但VGG网络和GoogleNet（代号也为Inception_v1）分别有19层和22层。

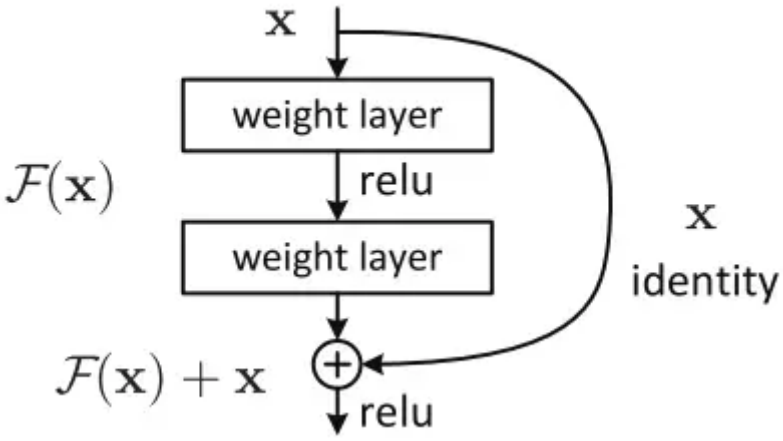
但是，如果只是通过简单地将层叠加在一起，增加网络深度并不会起到什么作用。随着网络层数的增加，就会出现梯度消失问题，这就会造成网络是难以进行训练，因为梯度反向传播到前层，重复乘法可能使梯度无穷小，这造成的结果就是，随着网络加深，其性能趋于饱和，或者甚至开始迅速退化。



增加网络深度导致性能下降

其实早在ResNet之前，已经有过方法来处理梯度消失问题，但遗憾的是，似乎没有一个方法可以真正解决这个问题。

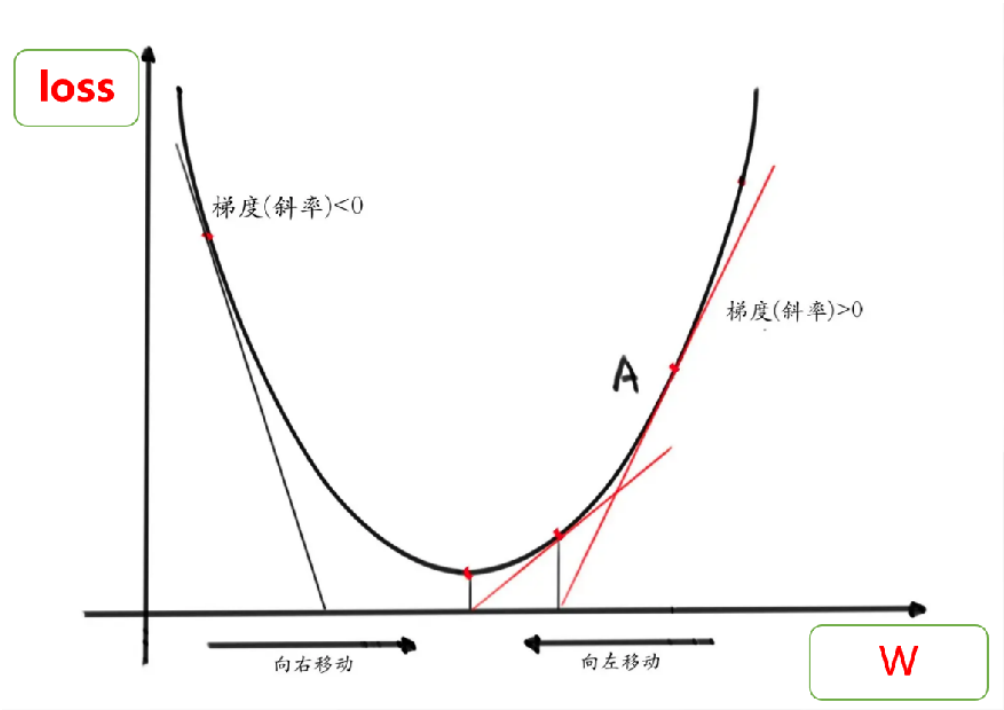
ResNet的核心思想是引入所谓的“恒等映射（identity shortcut connection）”，可以跳过一层或多层，如下图所示：



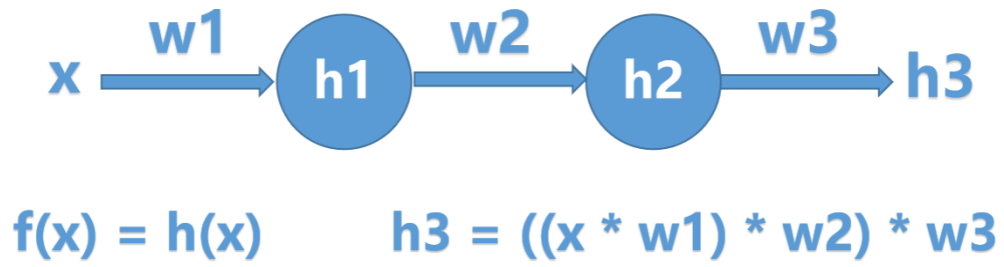
1.2 梯度消失问题分析

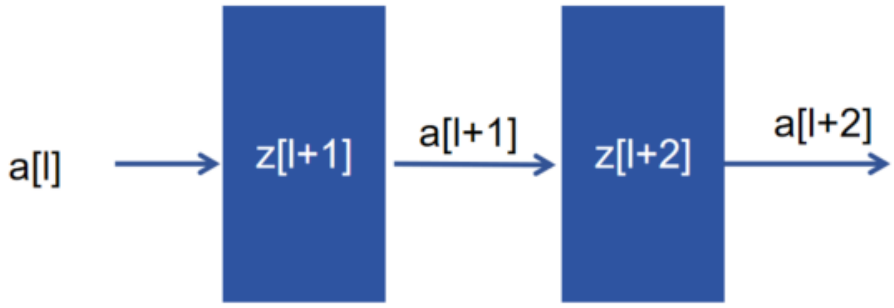
BP算法：链式求导（复合函数求导，导数连乘），连乘过程中(0.99^365)只要有一个值接近于0，整个乘法运算就接近于0，梯度则消失，模型失去了训练的意义。

loss：对loss进行链式求导，该导数根据梯度下降法进行更新权重。



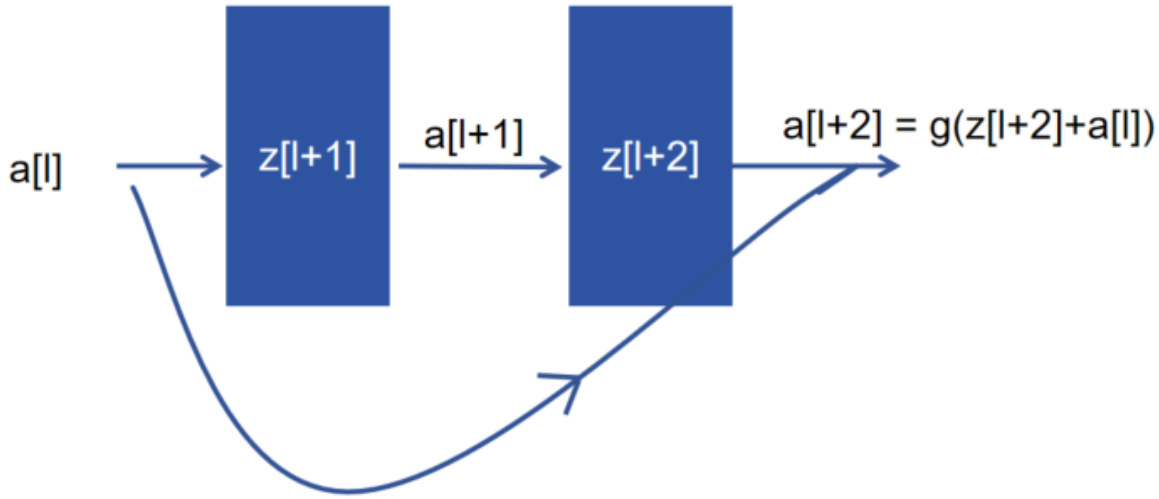
1.3 传统的连接方式






```
z[l+1] = w[l+1]a[l] + b[l+1]
a[l+1] = g(z[l+1])
z[l+2] = w[l+2]a[l+1] + b[l+2]
a[l+2] = g(z[l+2)]
a[l+2] = g(w[l+2]a[l+1] + b[l+2])
```

1.4 残差块的连接方式



在残差块中有一点变化，将a[l]直接向后拷贝神经网络的深层。意味着最后的等式a[l+2] = g(z[l+2]) 变成了 a[l+2] = g(z[l+2] + a[l]), 也就是加上的这个a[l]产生了一个残差块。

此时a[l]到a[l+2]的前向传播过程为:

```
z[l+1] = w[l+1]a[l] + b[l+1]
a[l+1] = g(z[l+1])
z[l+2] = w[l+2]a[l+1] + b[l+2]
a[l+2] = g(z[l+2] + a[l])
a[l+2] = g(w[l+2]a[l+1] + b[l+2] + a[l])
```

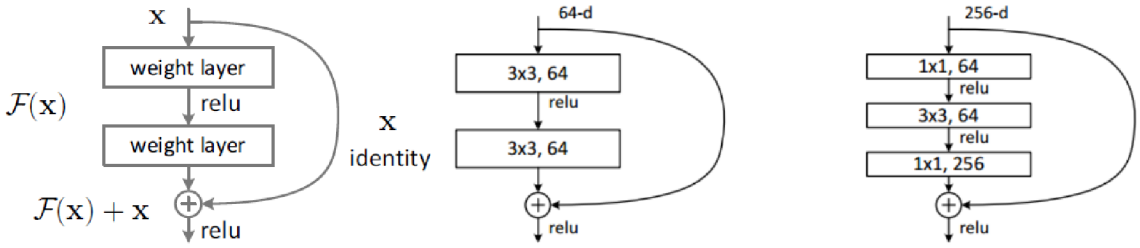
当前a[l] 就是残差块, 也有另一个术语叫做 **跳跃连接**, 指的就是a[l] 跳过一层或者几层，从而将信息传递到神经网络的更深层。

将a[l+2]展开，那么a[l+2] = g(W[l+2]a[l+1] + b[l+2] + a[l]), 假如当前W[l+2] = 0, 同时b[l+2]=0，那么残差的表达式就变为:

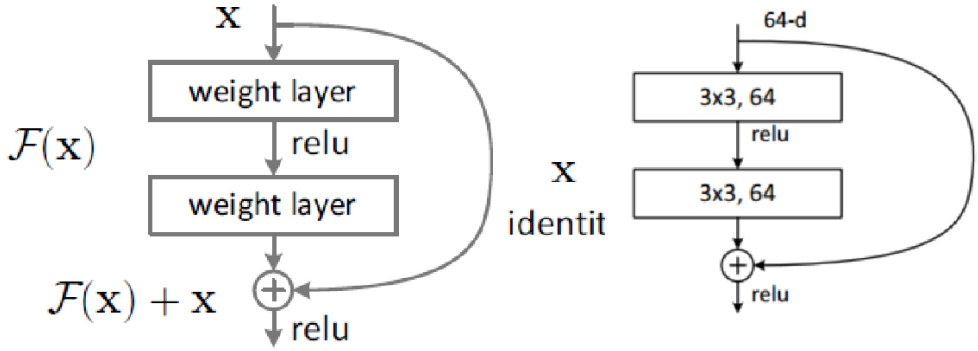
a[l+2] = g(a[l]) =a[l], 也就是获当出现参数为0的极端情况下，残差块也可以很容易得出a[l+2]=a[l], 意味着，即使神经网络增加了这两层，它的效果也不会差。因为恒等函数很简单。当然这是在极端的情况下，参数都为0，如果这些隐藏层学到一些有用的信息，那么它可能会比学习恒等函数表现都更好。

而那些不含有残差块的的普通网络情况就不一样，当网络不断加深，学习起来就会很困难。

1.5 残差网络结构



1.5.1基本结构

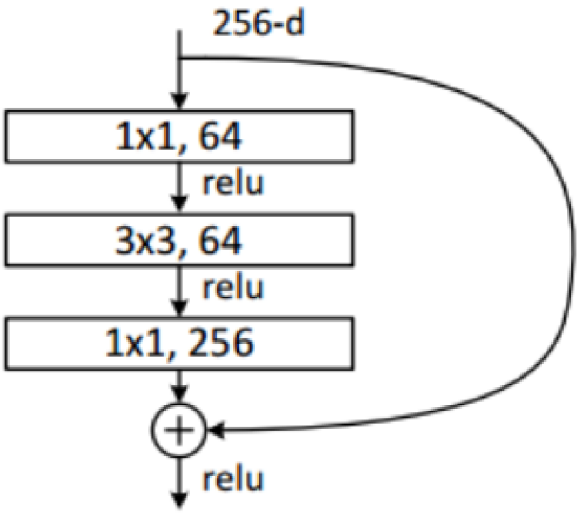



```
import torch
import torch.nn.functional as F
from torch import nn

class Res_Block(nn.Module):
    def __init__(self, c):
        super().__init__()
        self.conv1 = torch.nn.Conv2d(c, c, 3, padding=1)
        self.conv2 = torch.nn.Conv2d(c, c, 3, padding=1)

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

if __name__ == '__main__':
    x = torch.randn(1, 1, 32, 32)
    res_block = Res_Block(3)
    out = res_block(x)
    print(out)
    print(out.shape)
```

残差网络

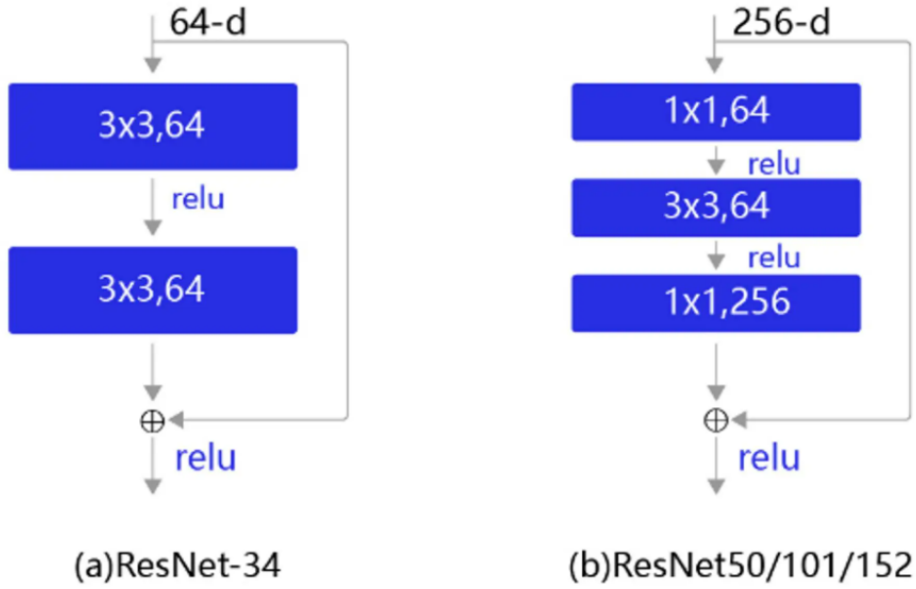
```python
import torch
import torch.nn.functional as F
from torch import nn


class Res_Block(nn.Module):
    def __init__(self, c):
        super().__init__()
        self.conv1 = torch.nn.Conv2d(c, c, 3, padding=1)
        self.conv2 = torch.nn.Conv2d(c, c, 3, padding=1)

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


class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = torch.nn.Conv2d(1, 16, 3, 1)
        self.rblock1 = Res_Block(16)
        self.conv2 = torch.nn.Conv2d(16, 32, 1, 1)

    def forward(self, x):
        out = self.conv1(x)
        x = self.rblock1(out)
        y = self.conv2(x)
        return y


if __name__ == '__main__':
    x = torch.randn(1, 1, 32, 32)
    # res_block = Res_Block(3)
    # out = res_block(x)
    net = Net()
    out = net(x)
    print(out)
    print(out.shape)
```

1.5.2 残差网络: 信息瓶颈



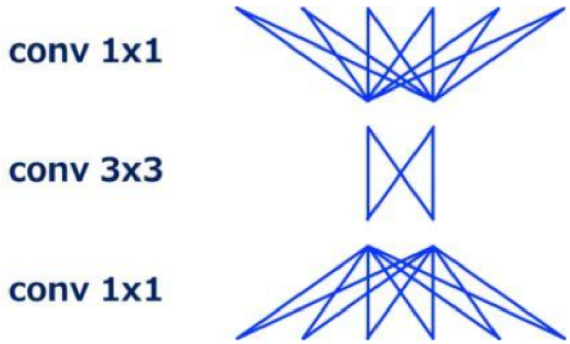
模型过滤噪声

信息瓶颈先降低通道，又还原通道，中间实现特征提取。效果类似于最大池化的去噪，所以较少与最大池化一起使用。



残差块是ResNet的基础，具体设计方案如上图所示。不同规模的残差网络中使用的残差块也并不相同，对于小规模的网络，残差块如 **图(a)** 所示。但是对于稍大的模型，使用 **图(a)** 的结构会导致参数量非常大，因此从ResNet50以后，都是使用 **图(b)** 的结构。

图(b) 中的这种设计方案常称作瓶颈结构（BottleNeck）。1*1的卷积核可以非常方便的调整中间层的通道数，在进入3*3的卷积层之前减少通道数（256->64），经过该卷积层后再恢复通道数(64->256)，可以显著减少网络的参数量。这个结构（256->64->256）像一个中间细，两头粗的瓶颈，所以被称为“BottleNeck”。



> 深度神经网络能够逐渐提取数据中的更高级别的特征，从而提高模型的泛化能力。bottleneck既增加了深度，计算量又可以得到节约。

简单计算一下残差块中使用 卷积前后参数量的变化，为了保持统一，我们令图中的两个结构的输入输出通道数均为256，

则图(a)中的结构所需的参数量为：

输入通道是256表示卷积核的通道数也是256, 一个卷积核的参数是3 * 3 * 256 ,因为输出通道是256 那么卷积核的个数是256

当前两个卷积核因此参数量为:

3 * 3 * 256 * 256 * 2=1179648

计算图b采用1*1卷积, 参数量为:

第一个卷积核1 * 1: 因为输入是256,因此卷积核通道数也是256, 输出通道数是64因此卷积核个数数64

第一个卷积核参数量: 1 * 1 * 256 * 64

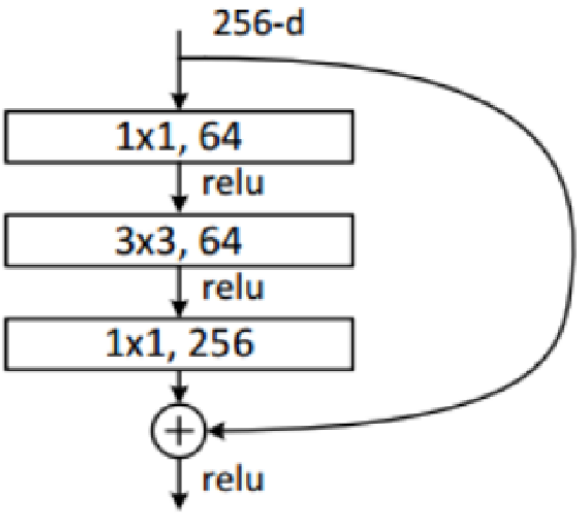
第二个卷积核参数量: 3 * 3 * 64 * 64

第三个卷积核参数量: 1 * 1 * 64 * 256

因此参数量加起来为:

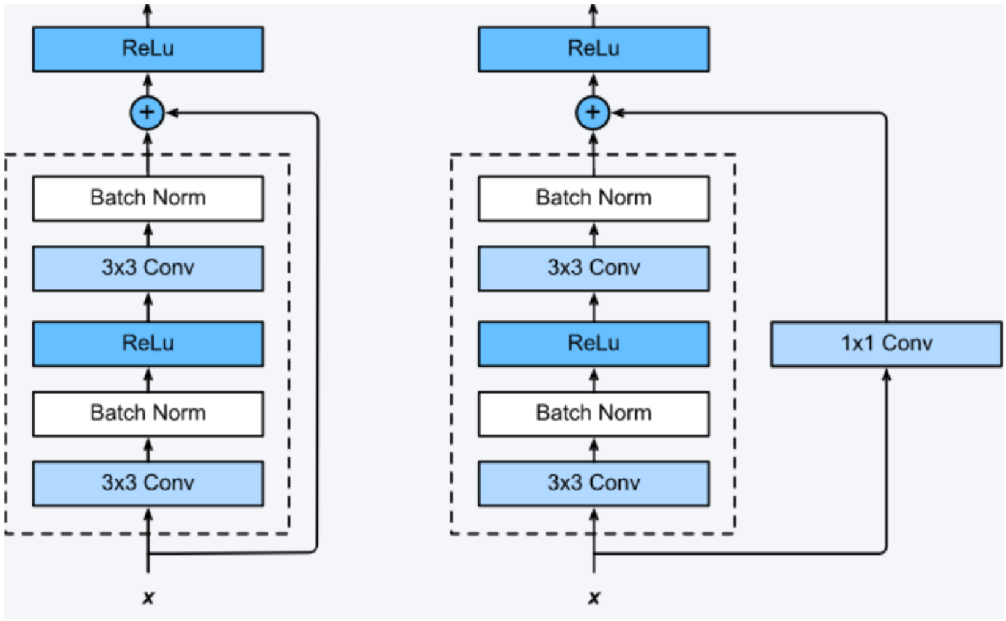
1 * 1 * 256 * 64 + 3 * 3 * 64 * 64 + 1 * 1 * 64 * 256 = 69632

代码实现:

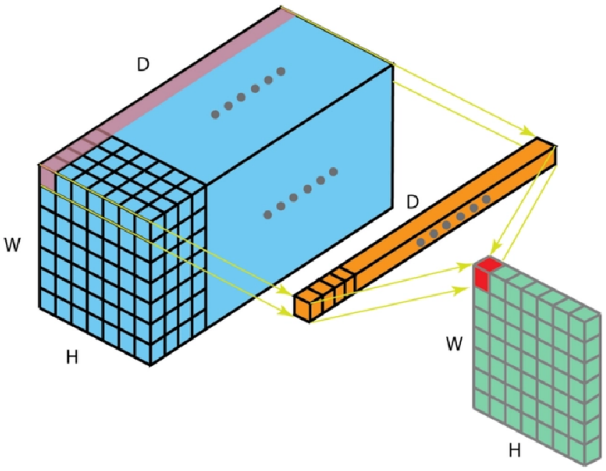



```python
import torch
from torch import nn
from torch.functional import F

class BottleneckBlock(nn.Module):
    def __init__(self, c_in, c_out):
        super().__init__()
        self.conv1 = nn.Conv2d(c_in, c_in//4, kernel_size=1)
        self.conv2 = nn.Conv2d(c_in//4, c_in//4, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(c_in//4, c_out, kernel_size=1)
    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = F.relu(self.conv2(out))
        out = self.conv3(out)
        # 添加残差
        out = F.relu(x + out)
        return out

if __name__ == '__main__':
    block = BottleneckBlock(256, 256)
    x = torch.randn(1, 256, 16, 16)
    out = block(x)
    print(out.shape)
```

2 残差网络: 1 * 1卷积核



1 * 1 卷积核作用



1. **降维**：1x1卷积核可以用来减少输入特征图的通道数，从而降低模型的计算复杂性。这种降维的效果有助于减小模型的参数量和计算开销。
2. **1x1卷积核升维（增加通道数）：** 通过另一个1x1卷积核，可以再次增加特征图的通道数，这一步升维有助于保持或增加网络的表达能力。这个升维的过程使得网络能够更好地捕捉和表示复杂的特征。
3. **引入非线性**：1x1卷积核中包含非线性激活函数，如ReLU（Rectified Linear Unit），可以引入非线性变换。这有助于模型学习更复杂的特征。
4. **增加模型深度提高非线性表示能力**
5. **跨通道的信息交融**：模型能够学习到不同通道之间的关系，捕捉到更丰富的特征表示。这在深度神经网络中尤为重要，因为不同通道之间可能存在复杂的相关性和非线性关系，通过1x1卷积核进行跨通道信息交融有助于提高网络的表达能力

3 梯度消失与梯度爆炸

链式求导法则：

链式求导法则是微积分中的一个基本原则，它描述了复合函数的导数计算方法。对于一个由多个函数组成的复合函数，链式求导法则规定了该复合函数的导数与各个组成函数的导数之间的关系。

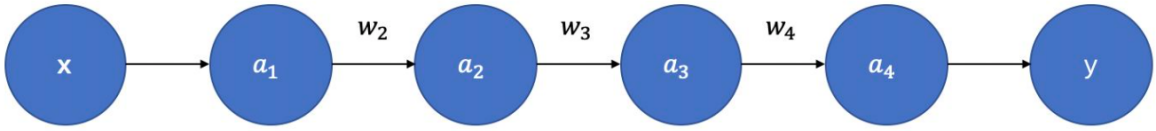
梯度消失：

梯度消失指的是在反向传播过程中，某些层的梯度变得非常小，甚至趋近于零。当网络很深时，由于链式求导法则的作用，梯度通过每一层都要相乘，如果这些乘积的结果接近零(0.99^365)，那么在更新网络参数时，这些层的权重几乎不会发生变化，导致模型无法学到有效的特征表示。

梯度爆炸：

梯度爆炸则是相反的问题，梯度在反向传播中变得非常大(1.01^365)。这可能导致权重更新过大，使模型不稳定，甚至导致溢出。

了解链式求导过程：



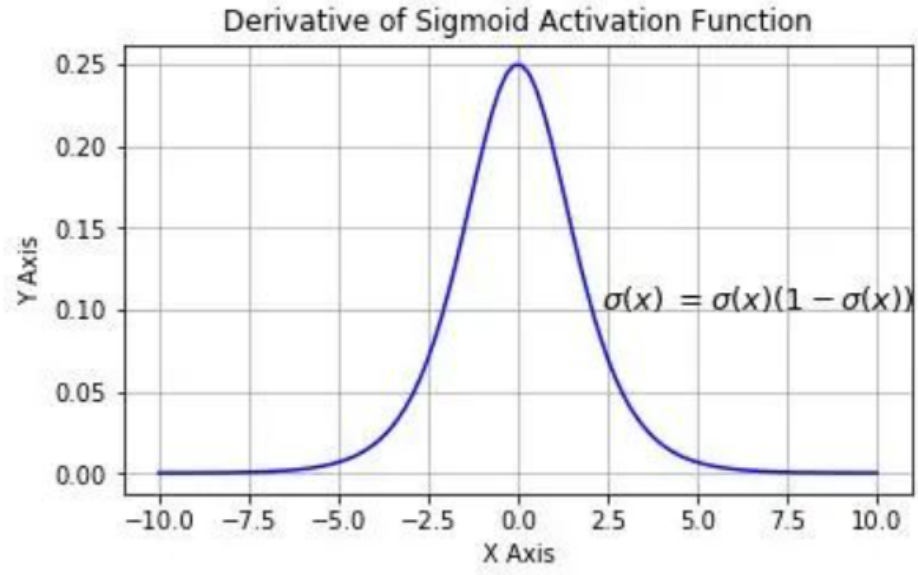
$$y_i = \sigma(z_i) = \sigma(w_i x_i + b_i)$$

```
σ是 sigmoid激活函数
```

ω1的梯度通过链式求导法则得出如下公式:

$$\sigma'(z_4)w_4\ \sigma'(z_3)w_3\sigma'(z_2)w_2\sigma'(z_1)$$

sigmoid导数图像下图所示



可以得到Sigmoid的导数最大值为0.25，随着网络层数的增加，小于1的小数不断相乘导致 逐渐趋近于零(0.99^365)，从而产生梯度消失。那么梯度爆炸又是怎么引起的呢？同样的道理，当权重初始化为一个较大值时，虽然和激活函数的导数相乘会减小这个值，但是随着神经网络的加深，梯度呈指数级增长(1.01^365)，就会引发梯度爆炸。从AlexNet开始，神经网络中就使用ReLU函数替换了Sigmoid，同时BN（Batch Normalization）层的加入，也基本解决了梯度消失或爆炸问题。

3.1 梯度消失与梯度爆炸

原因：网络层数太深，链式求导时的连乘效应会导致梯度爆炸或梯度消失。如果梯度值均小于1，则会出现衰减；如果都大于1，则会出现梯度爆炸。

梯度消失解决办法：

1、优化激活函数，使用relu(x大于0时，梯度为1)作为隐藏层激活，而sigmoid最大梯度为0.25，激活会导致梯度消失，只能作为输出层激活。

2、BN（BatchNormal）：把每一层神经网络的任意神经元输入值的分布规范为正态分布。

3、使用ResNet的跳线结构，避免梯度消失。

梯度爆炸解决办法

1. **梯度裁剪（Gradient Clipping）：** 通过设置一个阈值，当梯度超过这个阈值时，将梯度值缩放到合理范围内。这可以防止梯度值变得过大，从而稳定训练。
2. **权重初始化：** 合适的权重初始化方法可以减少梯度爆炸的风险。
3. **使用更稳定的激活函数：** 使用更稳定的激活函数，如ReLU（修正线性单元）或其变种，可以降低梯度爆炸的概率。
4. **减小学习率：** 降低学习率可以减缓权重更新的速度，从而降低梯度爆炸的风险。逐渐增加学习率或使用自适应学习率算法也是一种策略。

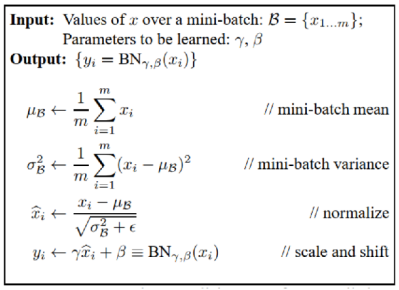
3.2 BatchNormal: 批归一化

数据批归一化：均值为 0 ，方差为 1 的正态分布。将输入数据映射到了原点周围，会导致激活函数结果产生变化，所以引入了缩放和平移，公式：

X：原始数据 **μβ：期望** **缩放** **平移**

$$y = \frac{x - mean(x)}{\sqrt{Var(x) + eps}} * gamma + beta$$

标准方差(误差的平均数) **增加计算稳定性**



训练深度神经网络很复杂，因为在训练过程中，每一层输入的分布会随着前一层参数的变化而变化。这样会训练带来什么问题?

主要体现在以下几个方面：

- **梯度消失和梯度爆炸**：当输入数据的分布发生剧烈变化时，网络层之间的梯度传播可能受到干扰，导致梯度消失（梯度变得非常小）或梯度爆炸（梯度变得非常大）问题。这可能导致网络难以训练或者训练变得非常缓慢。
- **训练不稳定性**：输入分布的变化可能导致训练的不稳定性，即模型在不同的迭代中表现不一致，难以收敛到稳定的结果。
- **收敛困难**：如果每一层的输入分布不稳定，网络可能需要更长的训练时间才能收敛到较好的结果。
- **过拟合**：输入分布的变化可能会导致模型在某些批次上过度拟合，从而影响模型的泛化能力。

为了应对这些问题，批归一化（Batch Normalization，BN）技术被引入。BN 可以在训练过程中标准化每一层的输入分布，使其保持稳定，从而缓解梯度消失和爆炸问题，加速收敛，提高训练效率，并增强模型的泛化能力。BN 层通过对每个小批量数据进行标准化和可学习的尺度调整，有助于使每一层的输入分布保持在一个适当的范围内，从而解决训练过程中的不稳定性问题。

通俗的对比讲解

批归一化（Batch Normalization，简称BN）操作，并用一个教小朋友的案例来进行通俗对比。
批归一化（Batch Normalization，BN）操作：
批归一化是一种在深度学习中广泛应用的技术，用于加速训练收敛、提高模型稳定性，以及防止梯度消失或梯度爆炸等问题。它的核心思想是在神经网络的每一层输入上进行归一化操作，以使输入保持在一个相对稳定的分布范围内。这有助于更快地训练网络，提高网络的泛化能力。

批归一化操作的步骤如下：
计算均值和方差： 对于每个小批量的数据，在每一层的输入上计算该批次数据的均值和方差。
归一化： 使用计算得到的均值和方差，对该批次数据进行归一化操作，将数据的分布调整为接近标准正态分布。
缩放和平移： 对归一化后的数据进行缩放和平移操作，通过可学习的参数来调整分布的形状，使得模型能够更好地适应不同的数据分布。
移动平均： 在训练过程中，随着每次迭代更新均值和方差的移动平均值，以便逐渐稳定归一化的效果。

教小朋友的案例对比：

想象你正在教一群小朋友跳绳，每个小朋友的技能水平和个子都不同。你希望他们都能够跳得好，但是要考虑到每个小朋友的个人差异。

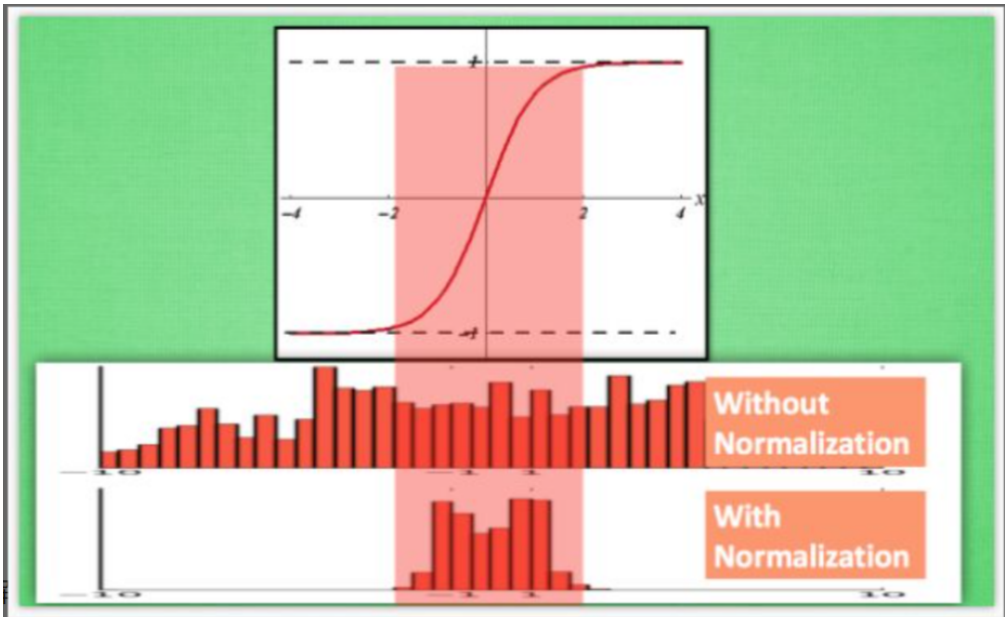
计算均值和方差： 在每次跳绳练习后，你观察每个小朋友跳绳的次数和速度，计算出每个小朋友的平均次数和速度。

归一化： 使用这些计算得到的平均次数和速度，对每个小朋友的跳绳次数和速度进行归一化，使得他们的表现在相对统一的范围内。
缩放和平移： 由于每个小朋友的个人特点不同，你可能需要根据他们的表现，对跳绳次数和速度进行适当的调整。一些小朋友可能需要多跳一些，一些小朋友可能需要加快速度。

移动平均： 随着每次练习，你会不断更新小朋友们的平均次数和速度，以便逐渐稳定他们的表现。

在这个案例中，你可以将你对小朋友的教学方式类比为批归一化操作。就像你为每个小朋友提供一个适合他们的训练环境，使得他们能够更好地学习跳绳一样，批归一化操作为神经网络的每一层提供了一个相对统一的输入分布，帮助网络更好地学习和适应数据的特征。

3.3 BatchNormal的作用



- 可以使用更大的学习率，提高训练速度
- 把bias设置为False

> 在使用Batch Normalization之后，可以将bias设置为False的原因是为了消除冗余。因为Batch Normalization层本身具有一个可学习的偏移参数，如果卷积层或其他层再设置bias，就会产生冗余

- 对权重初始化和大小不再敏感，权重初始化采样自0均值方差的正态分布
- BN可以加速深度神经网络的收敛速度。通过对每个小批量的输入进行归一化，使得网络在训练过程中更加稳定，减缓了梯度消失和梯度爆炸问题，使得学习更加快速有效。

> 使用sigmoid激活函数举例，从图中，我们很容易知道，数据值越靠近0梯度越大，值越大,梯度越接近0，我们通过BN改变数据分布到0附近，从而解决梯度消失问题。

> In traditional deep networks, too high a learning rate may result in the gradients that explode or vanish, as well as getting stuck in poor local minima. Batch Normalization helps address these issues. By normalizing activations throughout the network, it prevents small changes in layer parameters from amplifying as the data propagates through a deep network. For example, this enables the sigmoid nonlinearities to more easily stay in their non-saturated regimes, which is crucial for training deep sigmoid networks but has traditionally been hard to accomplish.
>
> 在传统的深度网络中，学习率过高可能会导致梯度爆炸或梯度消失，以及陷入差的局部最小值。批标准化有助于解决这些问题。通过标准化整个网络，在数据通过深度网络传播时，它可以防止参数的微小变化被放大。

- 不再依赖dropout，减少过拟合
- BN对每个特征通道进行归一化，使得网络对于不同尺度和幅度的输入更加鲁棒。

> “鲁棒性”表现在网络对于不同输入分布、不同尺度和幅度的输入数据更加稳定和适应，从而提高了模型的泛化能力和训练效果。

说明:

model.eval()转到测试阶段
模型使用之前训练好的平均均值和方差来进行归一化，而不是依赖当前小批量的统计信息。这有助于保持在训练和测试之间的一致性，确保模型在不同阶段的行为一致。

3.4 网络设计

```
self.conv_layer = nn.Sequential(
            nn.Conv2d(c, c, 3, 1, padding=1, bias=False),
            nn.BatchNorm2d(c),  # 在激活函数之前使用BN
            nn.ReLU()
)
```

卷积 BN 激活函数 池化

4 残差网络结构

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2 $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

代码:

```
import torch
from torch import nn, optim
import torchvision.transforms as transforms
from torchvision import datasets
from torch.utils.data import DataLoader
from torchvision.models import resnet18, resnet50, resnet101

train_data = datasets.CIFAR10(r'CIFAR10', True, transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((32, 32)),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
```

```
]), download=True)
train_loader = DataLoader(train_data, batch_size=128, shuffle=True)
test_data = datasets.CIFAR10(r'CIFAR10', False, transform=transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
]), download=True)
test_loader = DataLoader(test_data, batch_size=128, shuffle=True)
device = torch.device('cuda')
net = resnet18().to(device)
loss_func = nn.CrossEntropyLoss()
opt = optim.Adam(net.parameters())
for epoch in range(10000):
    net.train()
    for i, (img, label) in enumerate(train_loader):
        img, label = img.to(device), label.to(device)
        out = net(img)
        loss = loss_func(out, label)
        opt.zero_grad()
        loss.backward()
        opt.step()
    print(f"第{epoch}轮损失:{loss.item()}")
    net.eval()
    # 无需反向传播
    with torch.no_grad():
        sum_score = 0
        total_num = 0
        for img, label in test_loader:
            img, label = img.to(device), label.to(device)
            out = net(img)
            pre = out.argmax(dim=1)
            score = torch.eq(pre, label).float().sum().item()
            sum_score += score
            total_num += img.size(0)
        avg_score = sum_score / total_num
        print(f"第{epoch}轮得分{avg_score * 100}")
```

残差18代码:

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

class ResBlock(nn.Module):
    def __init__(self, c_in, c_out, s):
        super().__init__()
        self.s = s
        self.block = nn.Sequential(
            # 这里的步长需要改变
            nn.Conv2d(c_in, c_out, 3, stride=s, padding=1, bias=False),
            nn.BatchNorm2d(c_out),
            nn.ReLU(),
```

```
            # 这里的步长不需要修改
            nn.Conv2d(c_out, c_out, 3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(c_out),
            nn.ReLU()
        )

    def forward(self, x):
        if self.s == 1:
            return self.block(x) + x
        # 如果发生了下采样步长为2就不加残差
        else:
            return self.block(x)


class ResNet18(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_layer = nn.Sequential(
            nn.Conv2d(3, 64, 7, 2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.hidden_layer = nn.Sequential(
            # 这里没有下采样 conv2_x
            ResBlock(64, 64, 1),
            ResBlock(64, 64, 1),

            # 这里没有下采样 conv3_x
            ResBlock(64, 128, 2),
            ResBlock(128, 128, 1),

            # 这里没有下采样 conv4_x
            ResBlock(128, 256, 2),
            ResBlock(256, 256, 1),

            # 这里没有下采样 conv5_x
            ResBlock(256, 512, 2),
            ResBlock(512, 512, 1)
        )

        # 全连接
        self.out_layer = nn.Sequential(
            nn.Linear(1*1*512, 10)
        )

    def forward(self, x):
        out1 = self.input_layer(x)
        out2 = self.hidden_layer(out1)
        out2 = out2.reshape(-1, 1*1*512)
        out = self.out_layer(out2)
        return out

if __name__ == '__main__':
    x = torch.randn(2, 3, 32, 32)
```

```
    net = ResNet18()
    print(net)
    out = net(x)
    print(out.shape)
```