基于 python 的 21 层深度卷积神经网络

JY921127970142 网络空间安全

921127970142 汪哲男

联系方式： ericangean@163.com

189-0610-5703

摘要与声明

本文记录了基于python3.9实现的MNIST手写字迹识别**深度卷积神经网络**框架的搭构原理细节与经过。

深度网络模型共21层，由6个卷积层、7个ReLU激活层、3个池化层、2个Affine仿射层、2个Dropout层和1个Soft_with_Loss层组成。

使用的优化手段主要有：im2col()函数简化卷积计算、He 初始化权重参数、以AdaGrad最优化策略更新参数权重等，后文会有详细说明。

本文未使用任何现成的神经网络框架（如 tensorflow、pytorch 等），主要使用了NmuPy科学计算库与Matplotlib图形绘制处理库。使用了极少量他人代码，如im2col函数和col2im函数（修改自网络博客），后文中都会明确指出。

问题种类：分类问题，图像分类

921127970142 汪哲男

2022-12-15

21 层深度卷积神经网络

1 环境配置与数据处理

（1） Python 环境配置

分别官网下载 Python3.9、pip20.1 以及 PyCharm Community Edition 2022.3.exe。用 pip -list 确认当前 Matplotlib 已安装，如没有，则用 pip -install 命令安装。

（2） MNIST 数据集处理

MNIST 数据集分为 10 类，即 0~9 阿拉伯数字的手写体。原始数据都是 28*28 大小的黑白像素矩阵。

MNIST 图片示意图

上图展示了 MNIST 呈现 PNG 数据格式时是什么样，但为了便于之后的处理，我将直接对原始形式的MNIST数据集进行处理，将它转为便于Python处理的NumPy数组形式。

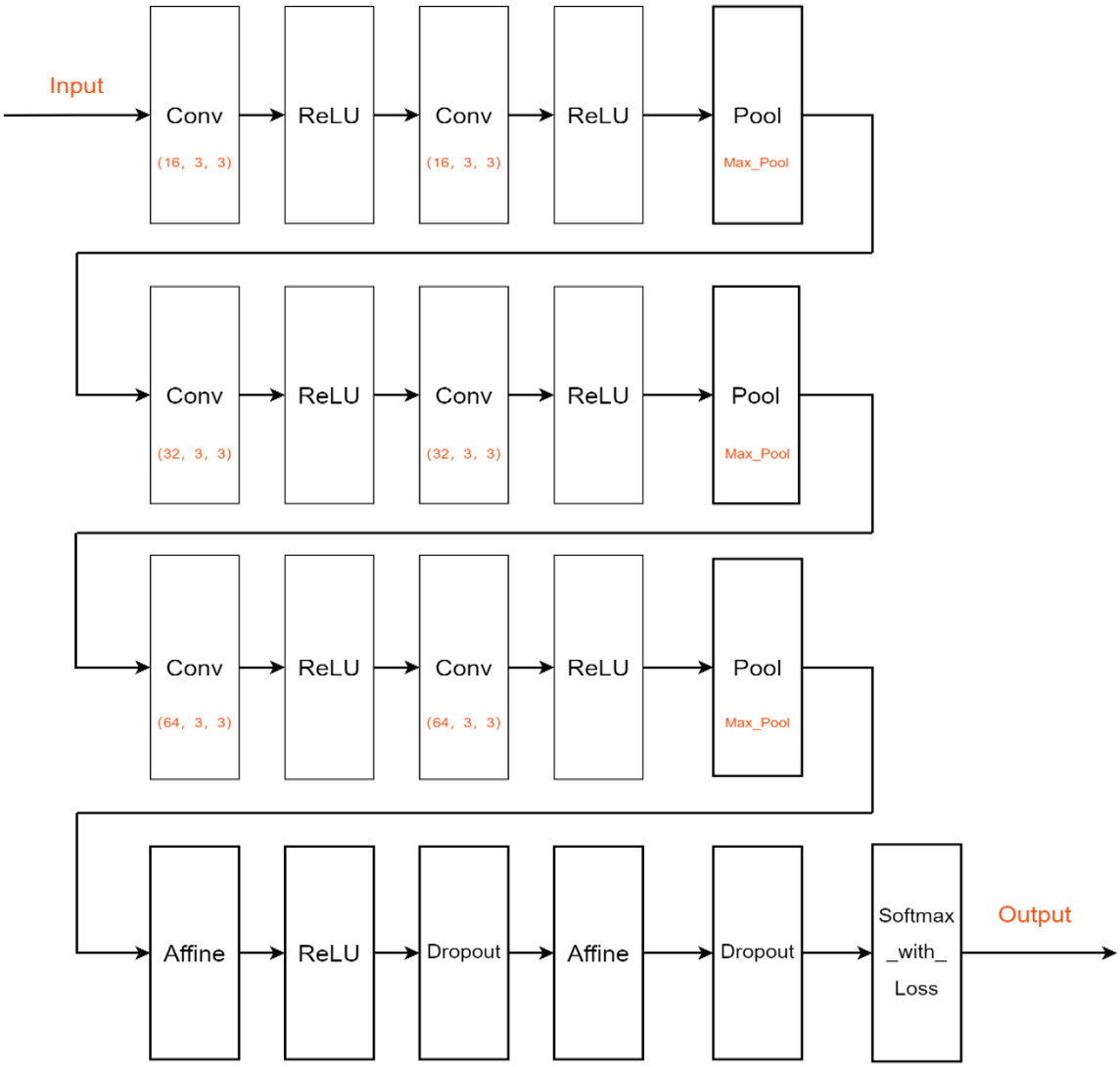
从官网 http://yann.lecun.com/exdb/mnist/ 下载了 MNIST 基本的 4 个.gz 文件后，运行如下代码，即可将 MNIST 数据集转化为 Numpy 数组形式并存储在 mnist.pkl 文件中

```
# key_file = {  储存着下载来的4 个MNIST 压缩包
#     'train_img':'train-images-idx3-ubyte.gz',
#     'train_label':'train-labels-idx1-ubyte.gz',
#     'test_img':'t10k-images-idx3-ubyte.gz',
#     'test_label':'t10k-labels-idx1-ubyte.gz'
# }
def load_img(file_name):
    file_path = dataset_dir + "/" + file_name
    with gzip.open(file_path, 'rb') as f:
            data = np.frombuffer(f.read(), np.uint8, offset=16)
```

```
    data = data.reshape(-1, img_size) %即 将列数固定为img_size，行数自动判断
    return data

def turn_to_Numpy(): %打好标签，储存到数组中
    dataset = {}
    dataset['train_img'] =  load_img(key_file['train_img'])
    dataset['train_label'] = load_label(key_file['train_label'])
    dataset['test_img'] = load_img(key_file['test_img'])
    dataset['test_label'] = load_label(key_file['test_label'])

    return dataset

def init_mnist():
    download_mnist()
    dataset = turn_to_Numpy()

    with open(save_file, 'wb') as f:
    %将Numpy数组储存到pickle文件里，方便以后直接调用
        pickle.dump(dataset, f, -1)
    print("Done!")
```

2 深度卷积神经网络架构

受经典神经网络 VGGNet 和 ResNet 启发，搭建了构造 6，3，7，2，2，1（6 个卷积层，3 个池化层，7 个 ReLU 激活层，2 个 Affine 仿射层、2 个 Dropout 层还有 1 个 Soft_with_Loss 层共 21 层的**深度网络**。



首先，使用 **He 初始化**方法初始化权重，并在后续过程中使用**基于 AdaGrad 的最优化策略**更新权重参数。

其次，正式开始学习与训练。将预处理后的图片以 28*28 像素大小的形式读入。所有的卷积层全都使用 3 × 3 的小型滤波器，且随着层的加深，通道数变大（卷积层的通道数从前往后依次为 16，16，32，32，64，64，以处理随着层数增加不断增加的数据）。以 ReLU 为激活函数。在卷积层中穿插池化层，用于压缩数据和参数的量，减小过拟合，同理在全连接层中使用 Dropout 层。最终层为 Softmax_with_Loss 层，此层输出预测结果与监督数据的差分，从而以此为凭据进行反向传播更新之前层的参数。

（以上内容将在第 3 部分进行具体论述）

卷积网络伪代码展示

Begin：

对数据进行预处理

初始化卷积核矩阵与偏置矩阵

使用 He 初始值作为权重初始值

Train：

If 迭代次数 > 预定次数 ：

Break

Else ：

读取待训练数据一条

CNNFP 前向转播：

#数据的表述形式 C*W*H 表示有 C 条 W*H 像素的数据

第一卷积层运算：1*28*28 -> 16*26*26

放入激活函数 Relu

第二卷积层运算：16*26*26 -> 16*24*24

放入激活函数 ReLU

第一池化层运算：16*24*24 -> 16*12*12

第三卷积层运算：16*12*12 -> 32*10*10

放入激活函数 ReLU

第三卷积层计算：32*10*10 -> 32*8*8

第二池化层运算：32*8*8 -> 32*4*4

第五卷积层运算：32*4*4 -> 64*4*4 #此处使用了大小为 1 的填充

放入激活函数 ReLU

第六卷积层运算：64*4*4 -> 64*4*4 #此处使用了大小为 1 的填充

放入激活函数 ReLU

第三池化层运算：64*4*4 -> 64*2*2

数据一维展开

放入全连接神经网络

Softmax_with_loss 层概率运算，输出分类概率结果

CNNBP 反向传播：

计算 softmax 函数局部梯度

计算全连接映射层局部梯度

修改全连接层权重与偏置

计算第三池化层局部梯度（获得多个矩阵）

计算第三卷积层局部梯度

修改第三卷积层的卷积核

计算第二池化层局部梯度（获得多个矩阵）

计算第二卷积层局部梯度

修改第二卷积层的卷积核

计算第一池化层局部梯度（获得多个矩阵）

计算第一卷积层局部梯度
修改第一卷积层的卷积核
参数清零
迭代次数加 1

TEST:

读取一条训练数据
将数据带入 CNN 神经网络
计算输出 10 个数字中，概率最大的下标
检查预测是否成功
计算成功率

核心参数、常数

```
1. lr = 0.01 #设置学习率初值
2. epochs = 8 #设置迭代次数
3. batch_size = 600 #设置训练样本数
4. evaluate_size = 1000 #设置测试样本数
5. filters1,filters2,filters3,filters4,filter5,filter6 = 16, 16, 32,
   32, 64, 64 #表示第1、2、3、4、5、6 卷积层各自的过滤器数量
6. filter_size1,filter_size2,filter_size3,filter_size4,filter_size5,f
   ilter_size6 = 3, 3, 3, 3, 3, 3 #表示第1、2、3、4、5、6 卷积层都使用
   3*3 尺寸的过滤器
7.
8. 正确标签 Y1~Y9 的 one-hot 表示：
9. Y1 = 1 0 0 0 0 0 0 0 0
10.Y2 = 0 1 0 0 0 0 0 0 0
11.Y3 = 0 0 1 0 0 0 0 0 0
12.Y4 = 0 0 0 1 0 0 0 0 0
13.Y5 = 0 0 0 0 1 0 0 0 0
14.Y6 = 0 0 0 0 0 1 0 0 0
15.Y7 = 0 0 0 0 0 0 1 0 0
16.Y8 = 0 0 0 0 0 0 0 1 0
17.Y9 = 0 0 0 0 0 0 0 0 1
```

注：一般情况下，epochs=8 batch_size=500 的训练规模无法使模型拟合。但本框架学习能力强、拟合速度极高，在 batch_size=500,lr=0.01 的情况下，只经过 4 轮迭代正确率就达到了 99%左右。因此出于节约计算资源考虑，将 epochs 和 batch_size 都定在小数字

3 框架各部分原理及代码实现

(1). 卷积层

在图像处理领域，卷积层较全连接层有着不可比拟的优势。原因是全连接层中数据的形状被“忽视”了。比如，输 入数据是图像时，图像通常是高、长、通道方向上的 3 维形状。但是，向全连接层输入时，需要将 3 维数据拉平为 1 维数据，隐藏在 3 维形状中的本质信息就被忽视了，造成了与形状相关信息的浪费。

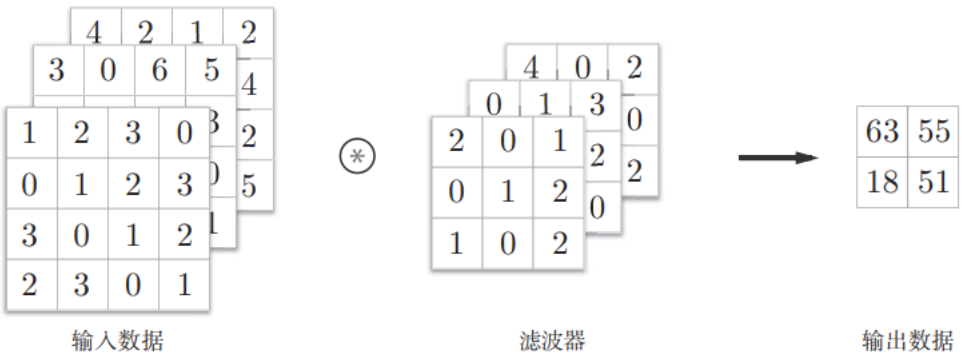
而卷积层可以保持形状不变。当输入数据是图像时，卷积层会以 3 维 数据的形式接收输入数据，并同样以 3 维数据的形式输出至下一层。因此，在 CNN 中，可以正确理解图像等具有形状的数据

卷积层的正向传播

正向的卷积运算中，卷积层以（H，W）的形式读入数据，H、W 分别表示高与宽。滤波器大小为（FH，FW），填充为 P，步幅为 S，输出大小(OH,OW)可由以下公式计算：

$$OH = \frac{H + 2P - FH}{S} + 1$$

$$OW = \frac{W + 2P - FH}{S} + 1$$



样例

代码实现：

实际训练时，由于卷积层其实需要同时处理 N 个数据，这 N 个数据又有不止两个维度，这让卷积运算的实现变得困难。为了降低实现难度，可以对卷积层的卷积运算进行如下优化。

我使用了一个名叫 im2col 的函数，即 image to column，翻译过来就是

“从图像到矩阵”的意思。Caffe,Chainer 等深度学习框架中都有此函数。

im2col 可以将三维的输入数据进行横向展开，形成一个个行向量。记每个输入数据为x_i,则输入数据最终形式为：

$$\boldsymbol{Input} = \left(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3} \ldots\ldots \boldsymbol{x_c}\right)^{\boldsymbol{T}}, \text{其中 } \boldsymbol{c} \text{ 为当前卷积层的通道数}$$

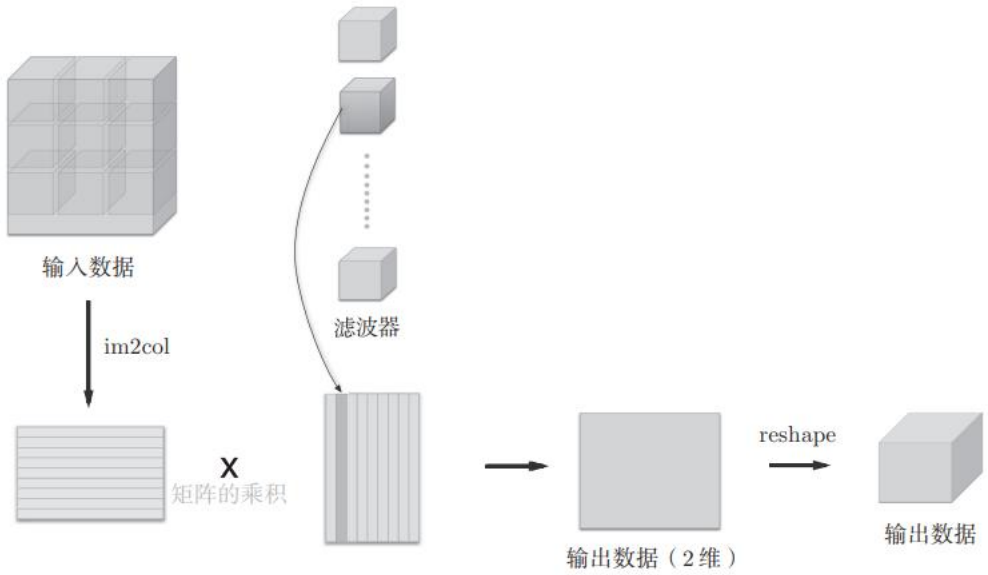
同时，im2col 将当前卷积层的若干过滤器进行纵向展开，形成一个个列向量。记每个过滤器为f_i，则最终过滤器的最终形式为：

$$\boldsymbol{Filter} = \left(\boldsymbol{f_1}, \boldsymbol{f_2}, \boldsymbol{f_3} \ldots\ldots \boldsymbol{f_s}\right), \text{其中 s 为当前卷积层的过滤器数}$$

由此，最终卷积层复杂的卷积运算就转变为了 Input 向量与 Filter 向量相乘的简单问题，输出为：

$$Output = Input \cdot Filter = \begin{pmatrix} x_1 f_1 & x_1 f_2 & \ldots & x_1 f_s \\ \ldots & & & \ldots \\ x_c f_1 & x_c f_2 & \ldots & x_c f_s \end{pmatrix}$$

图示如下：



代码如下：

```
1. #卷积层初始化
2. def __init__(self, W, b, stride=1, pad=0):
3.         self.W = W  #初始化权值
4.         self.b = b  #初始化偏置
5.         self.stride = stride  #初始化步长
6.         self.pad = pad  #初始化填充

  #正向传播代码：
1. def forward(self, x):
```

```
        FN, C, FH, FW = self.W.shape
        N, C, H, W = x.shape
        out_h = 1 + int((H + 2*self.pad - FH) / self.stride)
        out_w = 1 + int((W + 2*self.pad - FW) / self.stride)

        col = im2col(x, FH, FW, self.stride, self.pad)
        col_W = self.W.reshape(FN, -1).T

        out = np.dot(col, col_W) + self.b
        out = out.reshape(N, out_h, out_w, 1).transpose(0, 3, 1, 2
    )
        #读入时数据形式是(N,C,H,W)，但经过计算后已经变为了(N,H,W,C)，使
    用 transpose 函数将它变回来

        self.x = x   #保存中间数据，反向传播时要用到
        self.col = col
        self.col_W = col_W

        return out
```

注：im2col()的代码将在第 4 节给出

卷积层的反向传播：

卷积层的反向传播基于链式法则，即对于一个复合函数的求导，可以将它分解为若干个较简单的函数的求导的乘积。

在卷积层反向传播中，我们首先需要计算卷积层的损失函数对输出的梯度，然后再计算卷积层的损失函数对权重和偏置的梯度。

计算卷积层的损失函数对输出的梯度时，我们首先需要计算卷积层下一个层的损失函数对当前层输出的梯度。这个梯度可以通过链式法则得到。

然后，我们可以使用卷积的反向传播公式来计算卷积层的损失函数对权重和偏置的梯度。

最后，我们可以使用梯度下降算法来更新权重和偏置，以使损失函数最小化。

公式表述：

参数表：	
l	当前对应层数
L	网络总层数
δ	残差
⊙	哈达玛积
*	卷积
⊛	逆卷积，意为即δ^{l+1}执行普通卷积前需要以“FULL”模式进行填充值为 0 的 padding
·	乘
rot_{180}	表示矩阵旋转 180°

r 和 c 则是当前 δ 矩阵的行和列的索引，意即求 b 的梯度时对需要对 δ 进行空间维度上的求和。

$$\begin{cases} z^l = a^{l-1} * w^l + b^l \\ \delta^L = \nabla_{a^L} \cdot c \odot \sigma'(z^L) \\ \delta^l = (\delta^{l+1} \circledast rot_{180}\,(w^{l+1})) \\ \dfrac{\partial C}{\partial w^l} = a^l * \delta^l \\ \dfrac{\partial C}{\partial b_j^l} = \sum_{r,c} \delta_j^l \end{cases}$$

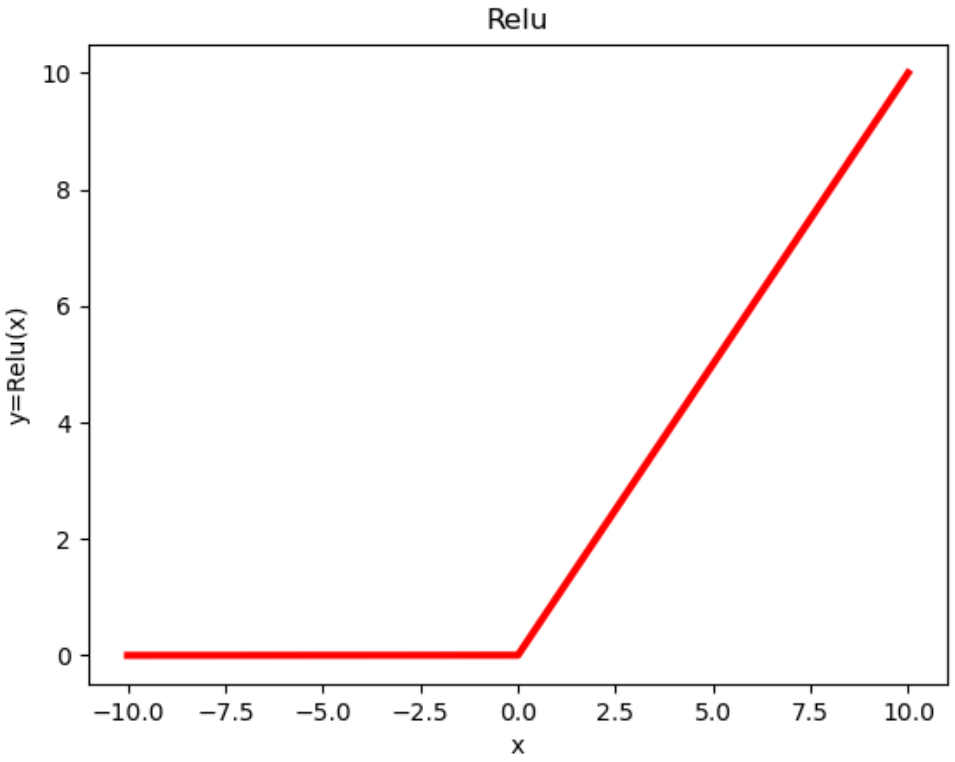
代码实现：

```
1. def backward(self, dout):
2.         FN, C, FH, FW = self.W.shape
3.         dout = dout.transpose(0,2,3,1).reshape(-1, FN)
4.
5.         self.db = np.sum(dout, axis=0) #计算偏置的梯度
6.         self.dW = np.dot(self.col.T, dout) #计算权重的梯度
7.         self.dW = self.dW.transpose(1, 0).reshape(FN, C, FH, FW)
8.
9.         dcol = np.dot(dout, self.col_W.T)
10.        dx = col2im(dcol, self.x.shape, FH, FW, self.stride, self.
   pad)
11.
12.        return dx
```

注：col2im()的代码将在第 4 节给出

(2). ReLU 激活层

ReLU 层中有 ReLU 函数，其函数图像如下：



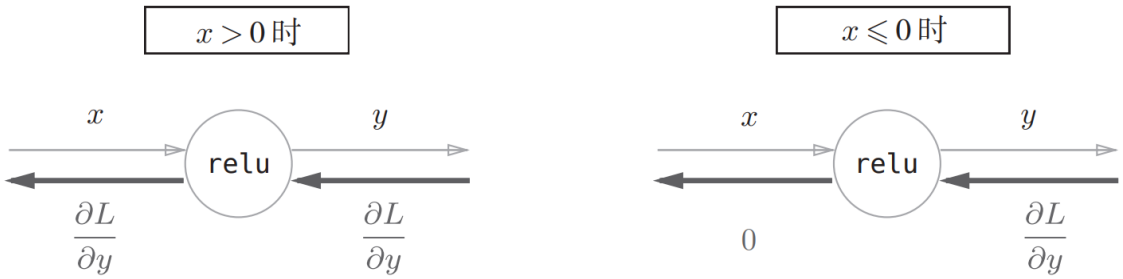
其数学形式可由下式表示：

$$y = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

容易求出 y 关于 x 的导数：

$$\frac{\partial y}{\partial x} = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

用计算图表示：



由以上公式，易得 RuLU 的正向/反向传播代码。

代码实现：

```
1. class Relu:
2.     def __init__(self):
3.         self.judge = None
4. #变量 judge 是由 1/0 构成的 Numpy 数组
```

ReLU 层正向传播：

```
1. def forward(self, x):
2.         self.judge = (x <= 0) #将输入 x 的元素中小于等于 0 的存为 1
3.                                 （True），大于 0 的存为 0(False)
4.         out = x.copy()
5.         out[self.judge] = 0
6.
7.         return out
```

ReLU 层反向传播：

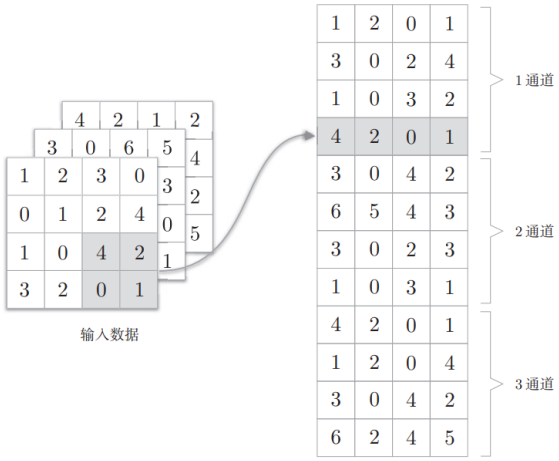
```
1. def backward(self, dout):
2.         dout[self.judge] = 0  #当 self.judge[i]为 1(True)即输入值小于
3.                               等于 0 时，第 i 个元素反向传播的值即为 0.
4.         dx = dout
5.         return dx
```

(3). MaxPool 池化层

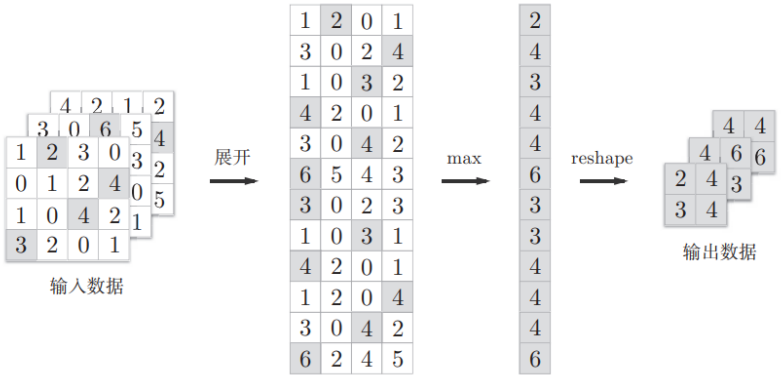
池化层可以在保留最本质信息的同时缩小训练数据的宽、高，从而提高学习效率。本框架池化层都采用了**宽高为(2,2)、步幅为 2** 的过滤器来进行 **Max 池化**。

MaxPool 池化层正向传播：

池化层正向传播的实现和卷积层相同，都使用 im2col 展开输入数据。不过，池化 的情况下，在通道方向上是独立的，这一点和卷积层不同。



如上图所示，像这样展开后，只需对展开的矩阵求各行的最大值，并转换为合适的形状即可。如下图所示：



代码实现：

```
1. def forward(self, x):
2.         N, C, H, W = x.shape
3.         out_h = int(1 + (H - self.pool_h) / self.stride)
4.         out_w = int(1 + (W - self.pool_w) / self.stride)
5.
6.         #改变数据形状，将连在一起的 2*2 方块展开成一个个行向量
7.         #行数由 reshape 自动判断
8.         col = im2col(x, self.pool_h, self.pool_w, self.stride, sel
   f.pad)
9.         col = col.reshape(-1, self.pool_h*self.pool_w)
10.
11.        #求出每一行最大的值，它们既是池化运算的结果，并将他们 reshape，
   方便进行下一次卷积运算
12.        arg_max = np.argmax(col, axis=1)
13.        out = np.max(col, axis=1)
14.        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2
   )       #将数据变回(N,C,H,W)
15.        self.x = x
16.        self.arg_max = arg_max
```

```
17.         return out
```

MaxPool 池化层反向传播：

池化层的反向传播和 ReLU 层原理类似。MaxPool 的前向传播是把 patch 中最大的值传递给后一层，那么**反向传播就是把梯度直接传给前一层某一个像素而其它像素不接受梯度。**

代码里，我在 MaxPool 的 forward()函数里专门计算了 arg_max，用来给 self.arg_max 赋值。**self.arg_max 变量是一个储存了 forward()运算中每一个最大值像素位置的 NumPy 数组。**借由 self.arg_max，可以轻松得出 MaxPool 池化层反向传播的代码。

代码实现:

```
1.     def backward(self, dout):
2.         dout = dout.transpose(0, 2, 3, 1)#将（梯度）数据还原成(N,H,W,C)格式
3.
4.         pool_size = self.pool_h * self.pool_w
5.         dmax = np.zeros((dout.size, pool_size))#生成一个空数组
6.         dmax[np.arange(self.arg_max.size), self.arg_max.flatten()] = dout.flatten() #把梯度以此加到 arg_max 记录的的位置上去
7.         dmax = dmax.reshape(dout.shape + (pool_size,))
8.
9.         dcol = dmax.reshape(dmax.shape[0] * dmax.shape[1] * dmax.shape[2], -1)
10.        dx = col2im(dcol, self.x.shape, self.pool_h, self.pool_w, self.stride, self.pad)
11.
12.        return dx
```

(4). Affine 映射层

Affine 层正向传播：

定义若干参数：

输入	$X(N,a)$	输入 X 是 N 个形状为(a,)的多维数组
权重	$W(b,c)$	权重 W 是一个形状为(b,c)的多维数组
偏置	$B(N,d)$	偏置 B 是 N 个形状为(b,)的多维数组

这样一来，神经元的加权和可以用公式

$$Y = X \cdot W + B$$

计算出加权和 Y。然后，Y 经过 ReLU 激活函数转换后，传递给下一层。这就是 Affine 层正向传播的流程。

代码实现:

```
1. def forward(self, x):
```

```
        # 对应张量
        self.original_x_shape = x.shape
        x = x.reshape(x.shape[0], -1)
        self.x = x

        out = np.dot(self.x, self.W) + self.b

        return out
```

Affine 层反向传播：

将 Affine 层之后的所有计算都记为函数 L，那么显然后一层向 Affine 层反向传播的导数是$\frac{\partial L}{\partial Y}$。又$Y = X \cdot W + B, \frac{\partial Y}{\partial X} = W, \frac{\partial Y}{\partial W} = X, \frac{\partial Y}{\partial B} = 0$,那么由链式法则易得

$$\begin{cases} \dfrac{\partial L}{\partial X} = \dfrac{\partial L}{\partial Y} \cdot W^T \\ \dfrac{\partial L}{\partial W} = X^T \cdot \dfrac{\partial L}{\partial X} \\ \dfrac{\partial L}{\partial B} = \dfrac{\partial L}{\partial Y} \text{在第 0 轴上的和} \end{cases}$$

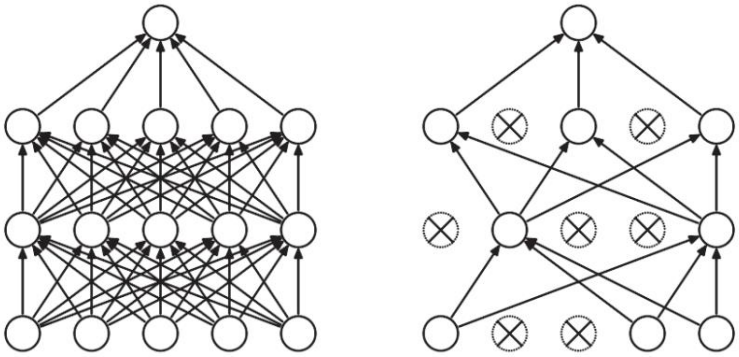
其中，W^T表示 W 的转置矩阵，X^T同理。

代码实现:

```
def backward(self, dout):
        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)

        dx = dx.reshape(*self.original_x_shape) #还原输入数据的形状
        return dx
```

(5). Dropout 层

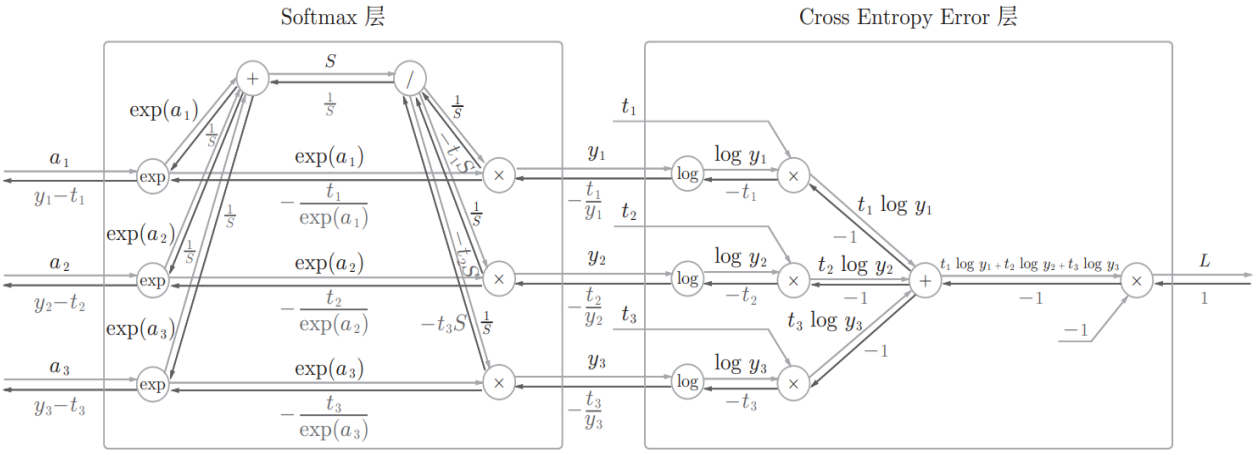
Dropout 是一种在学习的过程中随机删除神经元的方法。训练时，随机选出隐藏层的神经元，然后将其删除。被删除的神经元不再进行信号的传递。训练时，每传递一次数据，就会随机选择要删除的神经元。然后，测试时，虽然会传递所有的神经元信号，但是对于各个神经元的输出，要乘上训练时的删除比例后再输出。



Dropout 示意图，引自[1]：左边是一般的神经网络，
右边是应用了 Dropout 的网络

```
class Dropout:

    def __init__(self, dropout_ratio=0.5):
        self.dropout_ratio = dropout_ratio #设置删除系数
        self.mask = None #初始化 mask 数组，

    def forward(self, x, train_flg=True):
        if train_flg:
            self.mask = np.random.rand(*x.shape) > self.dropout_ra
tio
        #mask 随机生成和 x 形状相同的数组，并将值比 dropout_ratio 大的元素
        设为 True，否则为 False。False 即表示被删除。
            return x * self.mask
        else:
            return x * (1.0 - self.dropout_ratio)

    def backward(self, dout):
        return dout * self.mask
    #反向传播时，按原样传递信号；正向传播时没有传递信号的神经元，反向传
    播时信号将停在那里
```

(6). Softmax_with_loss 层



Softmax_with_loss 层可以拆分成 Softmax 层和 CEE 层（交叉熵误差层）两个子层，每层分别包含一个函数。(**Softmax 函数和 CEE 函数原理及代码将在第 4 节说明**，本节内容仅直接说明它们的效果及框架中的应用场景)

当 Soft_with_loss 层接收到一个输入信号$A = (a_{0,}a_1, \dots a_9)$时，Softmax 层首先将 A 正规化，得到$B = (b_0, b_1, \dots b_9)$。正规化后的$b_i \in [0,1]$,$且 \sum_{i=0}^{9} bi = 1$,此时的$b_i$可以神经网络认为原始图片上的数据是 i 的概率。

接着，CEE 层接受 Softmax 层的输出$(b_0, b_1, \dots b_9)$并将其拿来与监督标签$(t_0, t_1, \dots t_9)$进行差分，再从差分数据$(b_0 - t_0, b_1 - t_1, \dots b_9 - t_9)$中计算出误差 L。

最终 Soft_with_loss 层开启反向传播，利用误差 L 指导前面各层修改权重参数。

代码实现:

```
1. class SoftmaxWithLoss:
2.     def __init__(self):
3.         self.loss = None
4.         self.y = None # softmax 的输出
5.         self.t = None # 监督数据
6.
7.     def forward(self, x, t):
8.         self.t = t
9.         self.y = softmax(x)
10.        self.loss = cross_entropy_error(self.y, self.t)
11.
12.        return self.loss
13.
14.    def backward(self, dout=1):
15.        batch_size = self.t.shape[0]
16.        if self.t.size == self.y.size:
17.            dx = (self.y - self.t) / batch_size
```

```
        else:
            dx = self.y.copy()
            dx[np.arange(batch_size), self.t] -= 1
            dx = dx / batch_size

        return dx
```

4 优化及特殊函数说明

(1).已经说明过的优化

（1）.在全连接网络部分插入 Dropout 层减少过拟合
（2）.使用 im2col 函数和 col2im 函数优化卷积计算

(2).函数：im2col()及 col2im()

im2col 函数和 col2im 函数在上文已经说明过具体用法，在此不赘述。
值得说明的是，im2col()和 col2im()代码都非自己原创，皆是修改自网络博客

im2col 函数代码：

```
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    """

    Parameters
    ----------
    input_data : 由(数据量, 通道, 高, 长)的 4 维数组构成的输入数据
    filter_h : 滤波器的高
    filter_w : 滤波器的长
    stride : 步幅
    pad : 填充

    Returns
    -------
    col : 2 维数组
    """
    N, C, H, W = input_data.shape
    out_h = (H + 2*pad - filter_h)//stride + 1
    out_w = (W + 2*pad - filter_w)//stride + 1

    img = np.pad(input_data, [(0,0), (0,0), (pad, pad), (pad, pad)], 'constant')
    col = np.zeros((N, C, filter_h, filter_w, out_h, out_w))

    for y in range(filter_h):
```

```
        y_max = y + stride*out_h
        for x in range(filter_w):
            x_max = x + stride*out_w
            col[:, :, y, x, :, :] = img[:, :, y:y_max:stride, x:x_max:stride]

    col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N*out_h*out_w, -1)
    return col
```

col2im 函数代码：

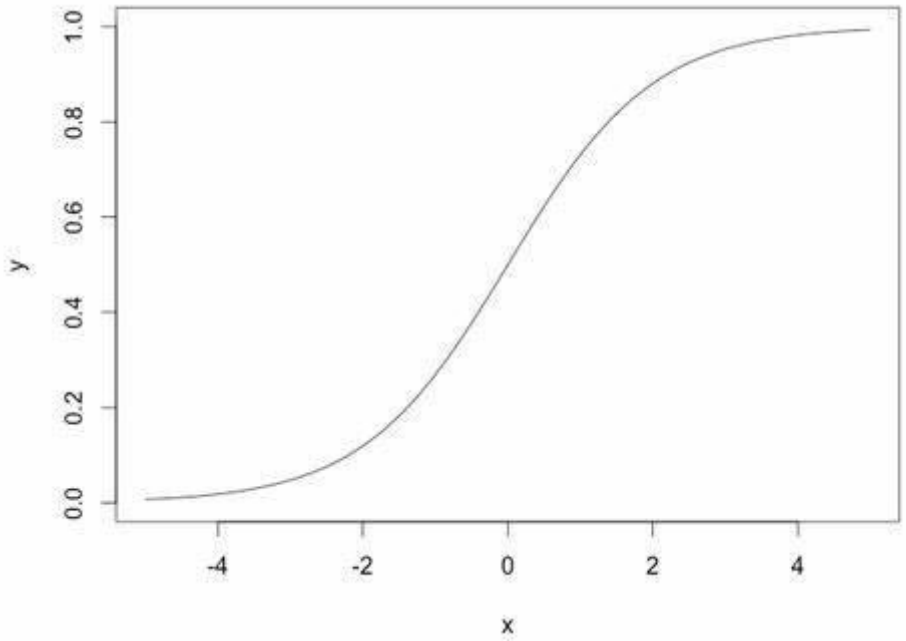
```
def col2im(col, input_shape, filter_h, filter_w, stride=1, pad=0):
    """

    Parameters
    ----------
    col :
    input_shape : 由(数据量, 通道, 高, 长)的 4 维数组构成的输入数据
    filter_h :
    filter_w
    stride
    pad

    Returns
    -------

    """
    N, C, H, W = input_shape
    out_h = (H + 2*pad - filter_h)//stride + 1
    out_w = (W + 2*pad - filter_w)//stride + 1
    col = col.reshape(N, out_h, out_w, C, filter_h, filter_w).transpose(0, 3, 4, 5, 1, 2)

    img = np.zeros((N, C, H + 2*pad + stride - 1, W + 2*pad + stride - 1))
    for y in range(filter_h):
        y_max = y + stride*out_h
        for x in range(filter_w):
            x_max = x + stride*out_w
            img[:, :, y:y_max:stride, x:x_max:stride] += col[:, :, y, x, :, :]

    return img[:, :, pad:H + pad, pad:W + pad]
```

(3).函数：Softmax

Softmax 函数输出的 Y 值是该项数据所属的类别可能性，数据所属的类别数据越高，该列属于当前类别的可能性越大。例如，输出的 Y 的值可能为(3,1,0.1,0.2,0,1,0.0,0.3)。可以直接从 Y 值中找到最大值，为 3，从而分到对应的类别，这在直观性和可操作性上差了很多。经过 softmax()函数最终的输出为概率，也就是说可以生成类似(86%,1%,5%,1%,3%,1%,1%,2%)的结果。其中 softmax()的公式如下：

$$S_i = \frac{e^i}{\sum_j e^j}$$

该公式可以分三步来说，一是以 e 为底对所有元素求指数幂；二时将所有指数幂数求和；三是分别将这些指数幂与该和做商。通过 softmax()函数处理之后输出结果的总和为 1，而且每个元素的结果可以代表概率值。Softmax()函数的图像如下图所示。



但是在计算机中 Softmax 还要面对溢出问题，因为 Softmax 函数要进行指数运算而指数函数的值可能会变得非常大。比如，e^{100}的值就有 40 多个尾 0，e^{1000}的结果回直接返回表示无穷大的 inf。如果在这些大数间就行除法运算，很可能会出现错误的值。

所以 Softmax 的基本公式需要改进：

$$S_i = \frac{e^i}{\sum_j e^j} = \frac{C \cdot e^i}{C \cdot \sum_j e^j} = \frac{e^{i+\log C}}{\sum_j e^{j+\log C}}$$

$$= \frac{e^{i+C'}}{\sum_j e^{j+C'}}$$

首先，在原式的分子和分母上都乘上常数 C（C 为任意常数），再将 C 移到指数函数

的幂上，记为$\log C$。最后将$\log C$替换为一个全新的符号C'。C'可以是任意常数，但为了防止溢出，我选择使用输入信号中的最大值。

Softmax 函数代码：

```
1. def softmax(a):
2.     c = np.max(a)
3.     exp_a = np.exp(a-c) #防止溢出
4.     sum_exp_a = np.sum(exp_a)
5.     s = exp_a / sum_exp_a
6.     return s
```

(4). 函数：Cross-Entropy-Error（CEE）

交叉熵误差函数（cross entropy error）这是一种适合分类任务的损失函数。该函数接收 softmax 的输出和监督标签，输出预测的损失误差。该函数的公式可以为：

$$\mathcal{L} = -\sum_{k} t_k log y_k$$

其中，y_k表示神经网络的输出，t_k则表示类别标签，k 表示数据的维度，log是以 e 为底数的自然对数。

Cross-Entropy-Error 函数代码：

```
1. def cross_entropy_error(y,t):
2.     if y.ndim == 1
3.         t = t.reshape(1,t.size)
4.         y = y.reshape(1,y.size)
5.
6.     batch_size = y.shape[0]
7.     return -np.sum(t * np.log(y+1e-7)) / batch_size
```

(5). 优化：He 初始化/Kaiming 初始化

He 初始化在网络使用 ReLU 激活时，可以使层激活的标准偏差维持在 1 左右，可以有效防止激活输出爆炸或消失，帮深度网络更快收敛。

$$w_i = \sqrt{\frac{2}{n_i}}$$

其中w_i表示第 i 层的初始权值；n_i表示第 i 层的神经元平均与前一层的几个神经元有连接（$n_i = C_{i-1} * W_{i-1} * H_{i-1}$）

He 初始化代码：

```
pre_node_nums = np.array([1*3*3, 16*3*3, 16*3*3, 32*3*3, 32*3*3, 64*3*3, 64*4*4, hidden_size])
wight_init_scales = np.sqrt(2.0 / pre_node_nums)
```

(6). 优化：用 AdaGrad 方法更新权重

AdaGrad（Adaptive Gradient Algorithm）的主要思想是：在各个维度上使用不同的 learning rate，从而加快模型的收敛过程。

$$\begin{cases} h = h + \frac{\partial L}{\partial W} \odot \frac{\partial L}{\partial W} \\ W = W - \mu \frac{1}{\sqrt{h}} \frac{\partial L}{\partial W} \end{cases}$$

W表示要更新的权重参数，$\frac{\partial L}{\partial W}$表示损失函数关于 W的梯度，μ表示学习率,h 保存了以前的所有梯度值的平方和

AdaGrad 代码：

```
class AdaGrad:
    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] += grads[key] * grads[key]
            params[key] -
  = self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

5 实验分析

模型中定义了一些超参，这些参数可以调节模型的层数以及神经网络的个数和训练时的一些设置(例如训练次数，学习率等)。下面将进行实验，探究一下主要超参数对结果的影响并在最后给出截图展示，首先我们将统一定义一下超参数的值为：

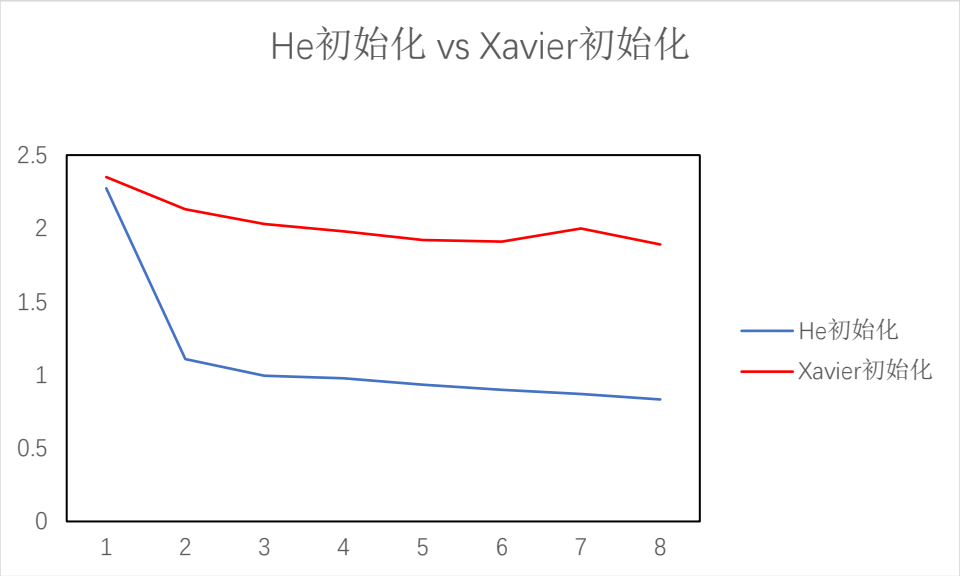
```
    lr = 0.01
    epochs = 8
    initialization_method ='He'#该参数其实并不存在
```

(1) 学习率与网络性能的关系

学习率	正确率
5e-2	49.55%
3e-2	97.86%
1e-2	99.30%
3e-3	98.38%
3e-5	65.07%
3e-7	17.95%

本次测试中 batch_size 和 epochs 依旧为 500 和 8 不变，分别设置不同的学习率看起效果。从以上的实验中可以发现当学习率在 0.01 到 0.003 之间时效果最好。

(2) 权重初始化方式与网络性能的关系

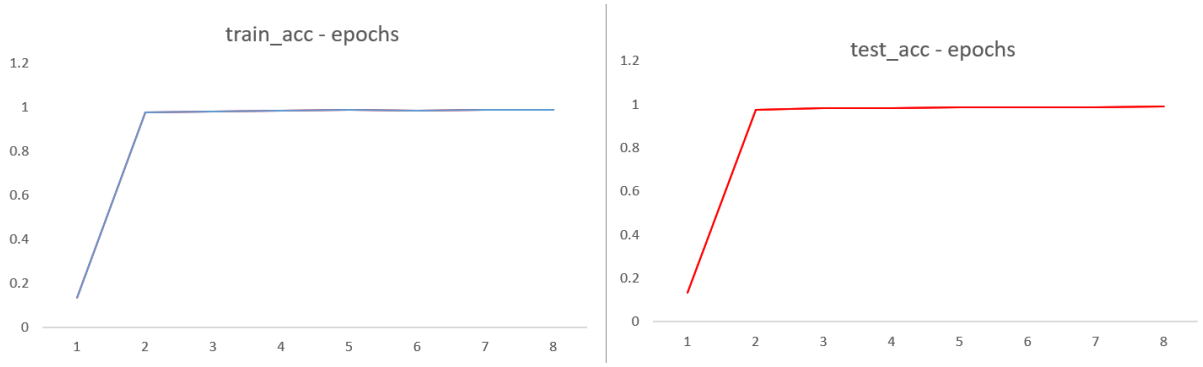


本次测试中 batch_size、epochs 和 lr 均保持不变，分别采取 He 初始化方法和 Xavier 初始化方法对权重参数进行初始化。分别训练后，将每个 epoch 中的平均训练 loss 制成折线图进行比较。

容易看到，虽然刚开始训练时两种初始化方法的模型误差保持在相近水平，但采用 He 初始化的模型在前 2 个 epoch 内误差就急剧减小，并在之后的 epoch 中稳定地持续降低误差；而 Xavier 初始化的误差在缓慢降低了 4 个 epoch 后则几乎稳定在了 1.9~2.0 这个区间内，表示此时网络已经几乎不再学习了。

综上所述，我认为可以得出结论，对于使用 ReLU 激活函数的深度网络，使用 He 初始化比 Xavier 初始化效果更好

（3）.模型准确率随训练次数的变化效果



本次测试中，保持 lr=0.01,batch_size=500 不变，使用了不同的训练次数，观察模型的准确率，可以看出来模型的准确率随训练次数不断提高，但训练次数在 5 次之后，效果提升不大。如图所示。

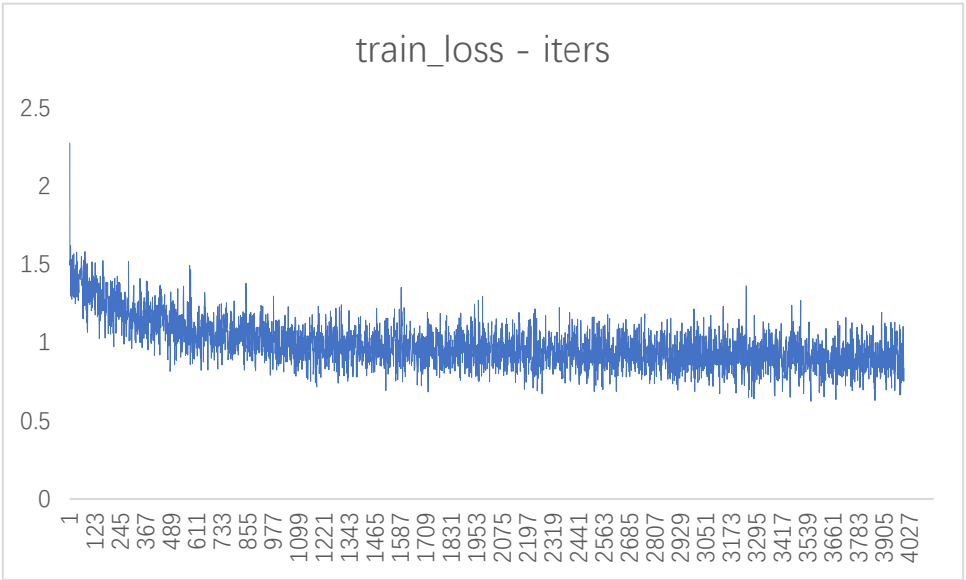
将训练 8 次时，前 8 次的平均准确率整理成表格：

训练次数	1	2	3	4	5	6	7	8
Train_acc	0.154	0.977	0.989	0.99	0.993	0.994	0.995	0.991
Test_acc	0.132	0.976	0.982	0.984	0.989	0.986	0.989	0.990
差值	0.022	0.001	0.007	0.006	0.004	0.008	0.006	0.001

可见，本深度学习模型能够在经历很少次数的学习、学习很少样本的前提下，快速达到极高的学习率，且未出现过度拟合。

我认为深度学习模型在解决 MNIST 手写数字识别问题上表现十分优异

（4）.学习速度随模型深度的变化效果



本次测试中，取 lr=0.01，epochs=8，batch_size=500 进行训练。将训练过程中的实时误差数据绘成图表。如图所示。

可见，头 1 个 epoch 的训练中误差骤然降低，之后的训练中误差整体持续降低并趋于稳定。

对此，我提出猜想，**是否模型越深，模型学习拟合的速度越快呢？**

于是，我在保持全连接层之后模型不变、超参数不变的条件下，改变在全连接层前卷积层激活层和池化层的数量，分别进行训练，并统计不同条件下前 3 轮 epoch 的平均误差

	第 1 轮	第 2 轮	第 3 轮
6 卷积 6 激活 3 池化	1.235361	1.052538	0.975401
5 卷积 5 激活 3 池化	1.436421	1.225461	1.123645
4 卷积 4 激活 2 池化	1.726161	1.465124	1.326121

由于，训练完 8 个 epoch 后，三个模型的准确率、误差都差不多，因此只挑误差变化最快的前 3 个 epoch 来看。

一目了然，6 卷积 6 激活 3 池化结构的模型学习速度最快。

综上所述，我推断，层数越深的模型学习收敛速度越快

附上几张运行截图：

```
train loss:0.9644045517545515
train loss:0.9132841859032481
train loss:1.0982647224811768
train loss:0.8469902504144742
train loss:0.8582562668876572
train loss:0.8933059965533094
train loss:0.9710202196186115
train loss:0.809735073426338
train loss:0.7575881328803032
train loss:0.8111311960027424
train loss:0.8936004482988741
train loss:0.9609360436088026
train loss:1.030682675150962
train loss:1.0302637029577941
train loss:0.9250590607900228
train loss:0.9620388722408597
train loss:0.9211128555123883
train loss:0.9118620020473429
train loss:0.9644126998005104
train loss:0.9871658573640311
train loss:0.9944989474297998
train loss:1.0480737852171076
train loss:0.9160071921253839
train loss:0.9541943974279684
train loss:0.9814331042246966
train loss:0.9384737247044338
=== epoch:8, train acc:0.99, test acc:0.992 ===
train loss:0.9214235067856302
train loss:0.8137782669632811
train loss:0.900495532314463
train loss:0.8623059380823568
train loss:1.0660190682241984
train loss:0.9757161982255813
train loss:0.9147914084111214
train loss:0.8217082261419411
train loss:0.9735158592966539
train loss:0.8494530338913063
train loss:0.9473548123985084
train loss:1.0344337030505606
train loss:0.9328138821528229
train loss:0.9495878835986553
train loss:0.8855978936296128
train loss:0.9182566775744274
train loss:0.825010271455666
train loss:0.8923973083022666
train loss:0.8395426223251411
train loss:0.9031297755329323
train loss:0.9889424404616927
train loss:0.9307682029993933
train loss:1.0256748920706722
train loss:0.8919572943778846
```

```
train loss:0.7692011611250458
train loss:1.032387437217052
train loss:0.9393964104839071
train loss:0.8512647097408572
train loss:0.906436437385873
train loss:0.957236892142018
train loss:0.9846313913287582
train loss:0.8052136838979096
train loss:0.9241010009524949
train loss:0.957796686633506
train loss:1.0452556891220615
train loss:0.8403927268462862
train loss:0.8473737140839167
train loss:0.9120874917212298
train loss:0.8765170255299877
train loss:0.9305161482280249
train loss:0.9158469756066546
train loss:0.9298207049396003
train loss:0.9926955571054736
train loss:0.8116836057294345
train loss:0.9894851289483378
train loss:0.8050698909083692
train loss:0.8439538596231556
train loss:0.878617447620199
train loss:0.9157251356334686
train loss:0.997634550050417
train loss:0.8741353515506393
=============== Final Test Accuracy ===============
test acc:0.9937
```