基于numpy的卷积神经网络手写数字识别

千朝 3220105001@zju.edu.cn

2024 年 7 月 20 日

目录

1 前言(README)

在人工智能的应用逐步全面普及的今天，对神经网络等人工智能基础知识的掌握正变得空前重要。

然而，在我大一自学卷积神经网络时，我尝试着在网上搜索过一些相关资料，却发现网络上大部分的资料都过于“简化”或原理化，而对于具体实现中的某些细节问题鲜有涉及。(例如，网上虽然有对卷积如何运算的简单讲解，但基本都局限于二维图像，很少有文章能讲清楚N*C*H*W的图像中N,C的含义，又如何将其变为N*O*H'*W'的结果等)这使我遇到了明白基本原理，却发现实际上的操作要比这一原理更复杂些，不知具体如何应用的问题，我相信或许有很多人遇到过和我类似的问题。

在修读《人工智能前沿》课程的时候，其实验任务是基于numpy实现一个LeNet结构的卷积神经网络，并用于手写数字识别任务。这给了我复习和梳理卷积神经网络相关知识的机会。

借着这一机会，我将这一课程的实验报告改编成了这篇文章，希望能为后来者对卷积神经网络的学习提供一些微小的帮助。

孟子说：“贤者以其昭昭，使人昭昭；今以其昏昏，使人昭昭。”检验一个人是否学懂某个知识的一个好方法就是看他能否把这一知识点给别人讲明白。所以，如果你发现你看不明白这篇文章的某些内容，那是因为我没有理解透或讲得不够好，我在此为其致歉，请不要因此而怀疑自己。如果你在这篇文章内发现了讲述不清晰的地方，欢迎反馈给我。

由于这篇文章的主体是课程实验报告，而实验报告是赶ddl快速写成的，难免可能存在一些纰漏。如果您发现这篇文章中有任何错误，欢迎向我反馈，感激不尽。

2 问题描述

使用神经网络来进行分类任务是深度学习的常见问题之一，而手写数字识别则是神经网络的入门级问题，常被用于深度学习领域的学习和理解。

MNIST是一个手写数字数据集，共包含70000个手写数字样本，其中60000个样本为数据集，其余的10000个样本为测试集。该数据集来自美国国家标准与技术研究所，其中训练集由来自250个不同人手写的数字构成，其中50%是高中生，50%来自人口普查局的工作人员，测试集也是同样比例的手写数字数据。MNIST被公开作为公共数据集，人和人都可以免费地获取和使用，是图像分类领域最常用的数据集之一。

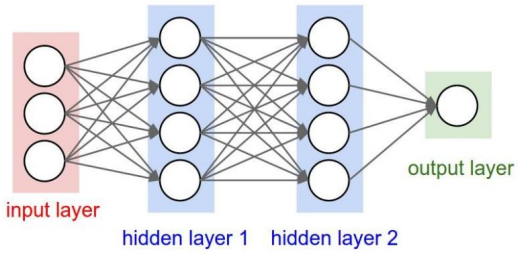
LeNet-5则是一种经典的卷积神经网络结构，最早被用于手写邮政编码字符识别，是图像分类领域的常用神经网络模型。LeNet-5由一个输入层、两个卷积层、两个池化层、两个全连接层、一个输出层组成(关于这些层具体是什么，我们将在下一部分介绍)，具有良好的学习和识别能力。

LeNet+MNIST手写数字识别是深度学习领域的经典问题，被称为深度学习的“Hello World”。在本次实验中，我们基于numpy实现LeNet-5并将其用于MNIST数据集的手写数字识别，并基于这一实验介绍卷积神经网络及其实现。

3 知识和思路整理

3.1 什么是神经网络

在搭建神经网络之前，我们首先得明白什么叫神经网络。下图是一张神经网络的示意图，其中红色为输入层，表示输入数据；蓝色为隐藏层[1]，表示神经网络内部对数据进行的一些处理和运算；绿色为输出层，表示神经网络运行后的输出结果。



图中的每一条箭头表示数据的传输方向，从输入层开始，每一个圆圈则表示一个函数，它对数据进行某种运算，然后再将数据传输至下一层进行下一步运算，最终得到输出结果(输出层)。需要注意的是，这张示意图中的隐藏层只有两层，但真正的神经网络中，隐藏层的层数一般会远多于两层；当然，输出层也可以不止一个数据(用于要同时预测多个量的情况)。

这张图就表示了神经网络中数据的传递和运算关系。我们可以看出，这张图及其表示的运算过程看上去与我们的神经系统较为相似(都是神经元间相互连接，逐步处理和传递信息)，所以我们将这种算法称为**神经网络**，而隐藏层中的每一个节点(圆圈)，我们称之为**神经元**。

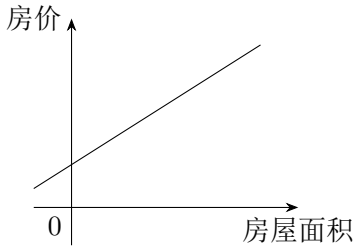
可以看出，神经网络其实就是某种由输入映射到输出的函数映射关系。下面，我们从线性回归模型开始，看看神经网络具体是怎样实现由输入到输出的映射的。

3.2 从线性预测开始

线性关系

我们从一个简单的预测问题开始入手:根据房屋面积预测房价。

一般来说，房子的价格等于面积乘上每平米房价，基本可以认为房价和房屋面积之间是线性关系。由于公摊面积和装修费等因素的存在，房屋面积和房价之间的函数关系可能不过原点，我们可以用一个一次函数来简单地基于房屋面积近似预测房价，如下图所示



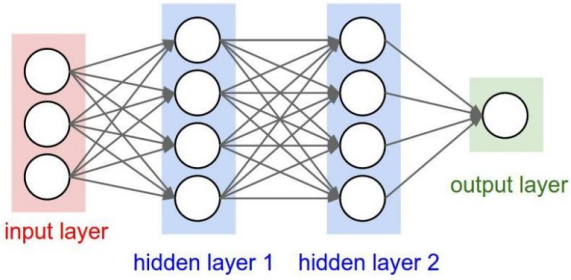
这一预测关系可以写成一个函数关系式$y = wx + b$

[1]隐藏层中的每一步运算是程序自动计算和运行的，我们只能看到输入输出，而看不到隐藏层中的这些运算过程，所以我们称其为“隐藏层”

更多自变量的引入-线性回归模型

在现实中，我们知道，房价不仅受房屋面积的影响，还与房屋地段、配套设施建设、小区绿化程度等多个因素有关。因此，在真正预测房价的时候，我们需要考虑多个变量的影响，而不是只基于房屋面积一个因素进行预测。

因此，我们需要建立一个模型，能基于多个自变量输入来进行运算得到预测输出。让我们回忆一下刚刚看到的神经网络的示意图，它是这个样子的



我们可以看到，这一模型就可以实现多变量输入来共同预测得到一个输出，能够满足我们的需求。不妨仍然考虑使用线性关系来进行预测，即假设每个神经元都是一个线性运算。对隐藏层(和输出层)每个神经元而言，如果来自上一层的数据分别记为$x_1, x_2, ..., x_c$的话，那么这种线性运算关系就可以表示为:

$$y = \sum_{i=1}^{c} w_i x_i + b$$

也可以用矩阵将其表示为

$$y = xW + b$$

你可能会疑惑为什么不写成$y = Wx + b$，我们将在后面对这一点予以说明。

像上面这个例子一样，如果神经网络中的每一个神经元都是线性的，我们就把这样神经网络称为**线性回归模型**。

不难发现，对于线性回归模型，不管中间有多少层隐藏层，由于中间的运算都是线性的，事实上最终的输出结果和输入仍是线性关系，所以线性回归模型即使按神经网络的结构来搭建，事实上仍然相当于一个$y = \sum_{i=1}^{c} w_i x_i + b$的线性变换(即可以由输入层经线性变换直接得到输出层)，没有必要做成神经网络的结构。但为了后续讲解的方便，我们仍将线性回归模型做成神经网络的结构来看待。

3.3 输出结果和损失函数

输出结果

对于输出结果(输出层)而言，既可以只输出一个值，也可以输出多个值。

- 单输出问题：对于房价预测这样的问题，我们只需要输出一个数字作为预测结果就好了，这是单输出问题。
- 多输出问题：我们考虑分类问题，理论上来说，我们也只需要给出一个输出，即预测的类别即可。但是更精准的，我们可能想要知道输入数据属于各个类别的概率。比如本次实验中我们需要

完成的手写数字分类，除了分类结果以外，我们可能希望神经网络可以更准确地告诉我们它判定输入图像是0-9中每个数字的概率。这样结果中就会含有10个概率值，这就是多输出问题。

损失函数

神经网络属于一种有监督学习。也就是说，当神经网络给出一个预测结果时，我们需要给出一个评价来衡量这一结果好不好，有多好(或者有多差)，它才能不断地向更好的方向学习。

为了给神经网络的预测结果一个评价，我们使用损失函数来衡量其输出结果与期望结果的偏差程度。显然，损失函数越小，表明神经网络的输出越好(离期望结果越接近)。

常用的损失函数有均方误差损失、Hinge损失函数、交叉熵损失函数等等。

在本次实验中，我们使用交叉熵损失函数来评判神经网络的预测效果。交叉熵是信息论中的一个概念，我们这里用交叉熵来构建损失函数评价神经网络的输出结果:

$$J = -\frac{1}{N}\sum a_i \log p_i$$

其中，N是数据量，a_i是正确结果，p_i是神经网络的输出结果。

因为损失函数越小，表明神经网络的输出越准确，所以神经网络的训练目标是使损失函数达到最小值。

3.4 前向传播和反向传播

神经网络的运行分为两部分——前向传播和反向传播，分别代表着从输入层开始向前运算到输出层得到结果的过程和从输出层开始反向更新参数优化神经网络的过程。下面我们来详细描述这两个过程是如何进行的。

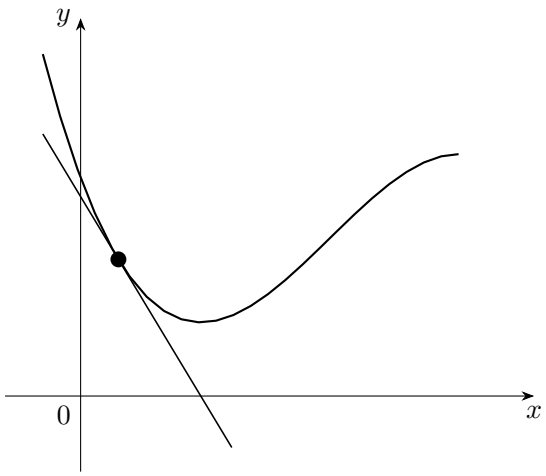
前向传播

当我们已经知道线性回归模型各神经元的w, b值时，就相当于已经知道了每个神经元的函数关系。此时我们只需要逐层计算，即可由输入层一层一层地向前计算，直到求出输出层的输出结果为止，这一过程即为前向传播。

前向传播的过程是简单的，简而言之只要各w, b值已知，让一路向前计算就好了。但是问题是，我们怎样寻找w, b值，让神经网络达到一个较好的预测效果呢?这就需要依靠反向传播了。

梯度下降法

在介绍反向传播之前，我们先介绍一下梯度下降法，这是一种寻找函数最小值的方法，也是反向传播和神经网络参数优化的基本原理。



让我们考虑上面图像中的这个函数，如果从一个随机的点出发(比如图中标出的点)，希望找到这个函数的最小值，我们应该怎么做呢?首先，我们作出图像的切线如图所示。我们看到，这条切线的斜率是负的，这表明x增大的方向y会减小，所以我们应该往x增大的方向移动一小段，然后再次作出切线，根据切线斜率的正负移动一小步，并重复上述步骤。当切线斜率为正时，我们应该往x减小的方向移动；当切线斜率为负时，我们应该往x增大的方向移动。反复执行这一操作，我们就可以逐步逼近这个函数的最小值。

这是一元函数的情况，而对于多元函数，类似地，如果要从某点出发寻找函数的最小值，我们只需要求出函数在这一点的梯度，然后往梯度的反方向移动一小步，再求梯度，再移动，以此类推，即可逐步逼近多元函数的最小值。

我们知道，梯度的反方向就是函数值减小最快的方向——这就像如何快速地从山上走到山谷?只需要沿着山坡最陡的方向往下走就可以了。这种方法就被称为梯度下降法。

梯度下降法可以表示为如下的迭代过程(其中的等号表示赋值):

$$\begin{aligned} &while\ looking\ for\ \min(f(\mathbf{x})):\\ &\qquad \mathbf{x} = \mathbf{x} - grad(f(\mathbf{x})) * lr \end{aligned}$$

其中lr是神经网络的学习率，决定了每次移动的一步是长还是短。

反向传播

梯度下降法告诉我们，只需要沿着梯度的反方向移动，就能找到函数的最小值(或是极小值)。而神经网络的训练目标就是达到损失函数的最小值。因此，我们只需要随机选取初始参数，然后一直往损失函数梯度的反方向移动就可以了。

问题在于，我们如何求出损失函数的梯度呢?

要求梯度，实质上就是求损失函数对各参变量的偏导数。我们首先要明确一点，因为我们优化的是各神经元的参数W和b，所以要求的也是损失函数对这些参数的偏导数$\frac{\partial J}{\partial W}, \frac{\partial J}{\partial b}$，而不是损失函数对输入数据的偏导数$\frac{\partial J}{\partial x}$

对于第l层的神经元，假设损失函数J对其输出结果$y^{[l]}$的偏导已经求出(上标$^{[l]}$表示这是一个第l层的

参数)，我们又知道该神经元的运算关系是

$$y^{[l]} = \sum_{i=1}^{c} w_i^{[l]} x_i^{[l]} + b^{[l]}$$

于是我们根据求导的链式法则可以知道，J对各参数的导数是

$$\frac{\partial J}{\partial w_i^{[l]}} = \frac{\partial J}{\partial y^{[l]}} \frac{\partial y^{[l]}}{\partial w_i^{[l]}} = x_i^{[l]} \frac{\partial J}{\partial y^{[l]}}$$

$$\frac{\partial J}{\partial b^{[l]}} = \frac{\partial J}{\partial y^{[l]}} \frac{\partial y^{[l]}}{\partial b^{[l]}} = \frac{\partial J}{\partial y^{[l]}}$$

于是我们就求得了损失函数J对参数W, b的偏导数，可以据此按梯度下降法更新参数W, b

进一步地，我们还可以求出J对这一层的输入$x^{[l]}$的偏导数

$$\frac{\partial J}{\partial x_i^{[l]}} = \frac{\partial J}{\partial y^{[l]}} \frac{\partial y^{[l]}}{\partial x_i^{[l]}} = w_i^{[l]} \frac{\partial J}{\partial y^{[l]}}$$

由于第l的输入实际上就是第$l-1$层的输出，所以我们实质上就求出了J对上一层各个神经元输出$y_i^{[l-1]}$的偏导数

$$\frac{\partial J}{\partial y_i^{[l-1]}} = \frac{\partial J}{\partial x_i^{[l]}} = w_i^{[l]} \frac{\partial J}{\partial y^{[l]}}$$

于是我们又可以对第$l-1$层神经元重复上面的梯度计算和参数更新过程了。

从上面的计算和分析可以看出，我们每计算出损失函数J对一层神经元各参变量的偏导数，就可以继续计算J对上一层神经元参变量的偏导数。以此类推，我们可以从损失函数对最后一层神经元的导数开始，逐层反向推出偏导数并进行参数更新。这一过程就被称为反向传播。更新的过程相当于(这里的等号表示赋值):

$$W = W - \frac{\partial J}{\partial W} * lr \qquad b = b - \frac{\partial J}{\partial b} * lr$$

神经网络运行的完整过程

对于训练集而言，神经网络要不断训练优化参数。其过程就是对训练集的每一个输入数据先前向传播得到输出并计算损失函数，再从输出层开始逐层反向传播求出损失函数对每一次神经元参数的偏导，再根据梯度下降法不断更新参数的。

对于测试集，神经网络只进行前向传播并输出预测结果，统计预测正确率，不进行反向传播更新参数。

可以看出，对神经网络的每一层而言，只要掌握其前向传播和反向传播，就能掌握整个神经网络的运行方式。

3.5 线性层(全连接层)

前向传播

在线性回归模型中，每一个神经元都是线性运算，公式表示为$y = xW + b$

如果某一层共有O个神经元，而上一层有C个神经元，则这一层每个神经元的运算可以表示为

$$y_1 = xW_1 + b_1$$
$$y_2 = xW_2 + b_2$$
$$\dots$$
$$y_O = xW_O + b_O$$

我们把这一层视作一个整体每个神经元的输出$y_1, y_2, \dots, y_O$作为一个向量y，那么这一层所有线性神经元的运算就可以简化地表示为:

$$y = xW + b$$

其中，x是C维向量，是这一层的所有输入构成的向量，y是O维向量，是这一层所有神经元的输出构成的向量。显然，b是O维向量，而W是$C \times O$的矩阵。

这样，我们就用一个式子表示了这一层中所有神经元的运算。因为这一层神经元执行的都是线性运算，所以我们把这样的层称为**线性层**。因为线性层接受上一层的所有输出作为这一层的输入，所以也被称为**全连接层**。

为了表示的方便和计算的简便，我们后面都以“层”为单位，对一整层进行运算和讨论，而不必对单个神经元进行分析。(你可能会留意到，我们把这一层神经元统一表示为一个$y = xW + b$的前提是这一层神经元全部都是线性的，如果其中有一个不是线性的，那就不能这样表示了。确实如此，但是正因为这一原因，为了运算和编程的方便，一般神经网络中一层神经元执行的都是相同类型的运算，所以可以以层，而不是以单个神经元，为单位进行表示)

在实际的神经网络训练过程中，有可能一次性输入多个数据作为一个批次(batch)来进行训练。假设一次性输入了N组数据的话，那么x应该是$N \times C$的，W仍然是$C \times O$的。b矩阵则由$1 \times O$的向量扩展成$N \times O$的矩阵(“扩展”的意思是，这个$N \times O$矩阵的N行都是一样的，即每一行都是原本那个$1 \times O$向量，这和numpy的广播机制[2]是一致的)。

这也是我们为什么要写成$y = xW + b$而不是$y = Wx + b$的原因，对于$y = Wx + b$，x是$N \times C$的，而y是$N \times O$的，那么Wx显然无法保持与y尺寸相同——所以$y = Wx + b$的表示方法不合适。

总之，线性层的前向传播公式是

$$y = xW + b$$

x - shape(N,C)
W - shape(C,O)
b - shape(1,O)，扩展为(N,O)
y - shape(N,O)

反向传播

对第l层，我们求其反向传播的公式。我们需要根据$\frac{\partial J}{\partial y}$求出损失函数$J$的偏导$\frac{\partial J}{\partial W}, \frac{\partial J}{\partial b}$用于参数更新，并需要求出$\frac{\partial J}{\partial x}$用于进一步的反向传播。

我们已知线性层的运算是$y = xW + b$，于是有

[2]鉴于这是《人工智能前沿》的实验报告，而不是python课程的，关于numpy的知识这里就不详述了

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial J}{\partial y}W^T$$

$$\frac{\partial J}{\partial W} = x^T\frac{\partial J}{\partial y}$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} = \frac{\partial J}{\partial y}$$

其中，$\frac{\partial J}{\partial x}, \frac{\partial J}{\partial b}$的推导是简单的，而$\frac{\partial J}{\partial W}$的推导则较为复杂。由于篇幅限制，这里不详细推导了，感兴趣的读者可以查看这篇文章。

对于N张图作为一个batch一起输入的情况，我们求导数时单纯地把每张图的导数相加即可。

3.6 向非线性环节出发

非线性影响因素的存在

我们前面的讨论局限于线性回归模型，它最大的特点就是输出和输入之间的关系是线性的。但是，在实际生活中，大部分变量之间的关系其实是非线性的，如速度和空气阻力之间的关系、半长轴长和行星公转周期之间的关系等等。如果用线性回归模型对这些非线性关系来进行预测，可以想象到，其结果将会有很大的误差。

因此，为了能更好地预测非线性变量之间的关系，我们需要在神经网络中引入一些非线性的函数关系，使其从线性回归模型变为真正的非线性的神经网络。

非线性神经元的形式

既然我们要将神经网络变为非线性的，那么就需要将线性神经元变为非线性神经元。

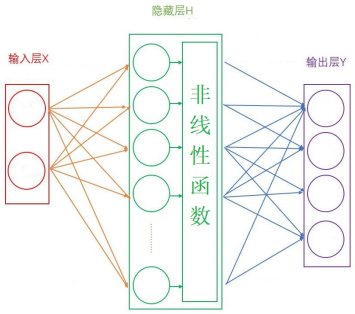
同时，我们还希望每一层非线性层的输出都能基于这一层的所有输入来得到，即希望非线性层也是“全连接”的，而全连接是线性层的特点。

所以，我们在神经网络中常用的非线性神经元一般是线性函数+非线性函数这样的形式，即

$$y = f\left(xW + b\right)$$

其中$f(x)$是非线性函数。

这样的一层神经元也可以理解为一层线性神经元和一层非线性神经元的叠加，可以用下面这张图来理解这种结构



其中，隐藏层中的圆形表示线性神经元。从图中可以看出，数据到达非线性的隐藏层时，先经过线性神经元运算得到$xW+b$，再经过一个非线性函数得到该隐藏层的输出$f(xW+b)$

对于函数$f(x)$，当其自变量为矩阵时，则表示对矩阵中的每一个元素分别计算其函数值$f(\cdot)$，即

$$f(A)=\begin{bmatrix} f(a_{11}) & f(a_{12}) & \dots & f(a_{1n}) \\ f(a_{21}) & f(a_{22}) & \dots & f(a_{2n}) \\ \vdots & \vdots & \ddots & \vdots \\ f(a_{n1}) & f(a_{n2}) & \dots & f(a_{nn}) \end{bmatrix}$$

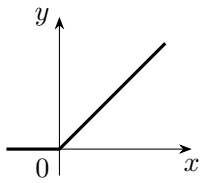
3.7 激活函数

非线性神经元$f(xW+b)$中的$f(x)$被称为激活函数，下面介绍几种常用的激活函数。

ReLU

ReLU函数的表达式是

$$ReLU(x)=\begin{cases} x & ,x>0 \\ 0 & ,x\le 0 \end{cases}$$



其优点是结构简单、计算快，并且因为$x\le 0$时函数值恒为0，可以忽略某些不想关的输入数据x

其缺点是有时部分相关数据也可能会落入$x\le 0$区间被忽略，造成神经元死亡问题。

其反向传播(导数)公式为:

$$\frac{\partial J}{\partial x}=\frac{\partial J}{\partial y}\frac{\partial y}{\partial x}=\begin{cases} \dfrac{\partial J}{\partial y} & ,x>0 \\ 0 & ,x\le 0 \end{cases}$$

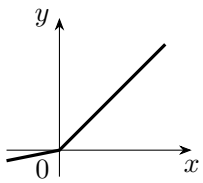
数学上，其导数在$x=0$处是没有定义的，但在实际运用中，为了方便，不妨将其视作0处理

Leaky ReLU

Leaky ReLU函数是ReLU函数的一个改进版，它在一定程度上缓解了ReLU函数可能造成神经元死亡的缺点，其表达式为

$$Leaky-ReLU(x)=\begin{cases} x & ,x>0 \\ ax & ,x\le 0 \end{cases}$$

其中$0<a<<1$



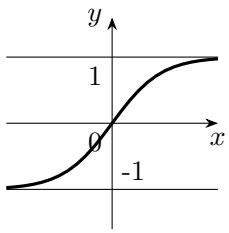
其反向传播公式为:

$$\frac{\partial J}{\partial x}=\frac{\partial J}{\partial y}\frac{\partial y}{\partial x}=\begin{cases}\dfrac{\partial J}{\partial y} & ,x>0\\ a\dfrac{\partial J}{\partial y} & ,x\leq 0\end{cases}$$

tanh

双曲正切函数$\tanh(x)$也是一个常用的激活函数，其表达式为

$$\tanh(x)=\frac{e^x-e^{-x}}{e^x+e^{-x}}$$



其优点是中心对称，分布在(-1,1)之间；缺点是设计指数的运算，计算较慢

我们计算其导数:

$$\tanh'(x)=\frac{4}{(e^x+e^{-x})^2}=1-\frac{e^{2x}+e^{-2x}-2}{(e^x+e^{-x})^2}=1-\left(\frac{e^x-e^{-x}}{e^x+e^{-x}}\right)^2=1-(\tanh(x))^2$$

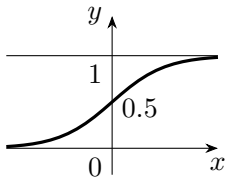
因此，其反向传播公式为

$$\frac{\partial J}{\partial x}=\frac{\partial J}{\partial y}\frac{\partial y}{\partial x}=(1-y^2)\frac{\partial J}{\partial y}$$

Sigmoid

Sigmoid函数在神经网络中也常被用作激活函数，其表达式为

$$\text{Sigmoid}(x)=\frac{1}{1+e^{-x}}$$



其优点是分布在(0,1)之间，可以用于归一化，并且关于$(0,f(0))$，即$(0,0.5)$对称；与tanh类似，其缺点是涉及指数运算，计算较慢。

我们计算其导数:

$$\text{Sigmoid}'(x)=\frac{e^{-x}}{(1+e^{-x})^2}=\frac{1}{1+e^-x}\cdot\frac{e^{-x}}{1+e^{-x}}=\text{Sigmoid}(x)\left(1-\text{Sigmoid}(x)\right)$$

因此，其反向传播公式为

$$\frac{\partial J}{\partial x}=\frac{\partial J}{\partial y}\frac{\partial y}{\partial x}=y(1-y)\frac{\partial J}{\partial y}$$

3.8 卷积层

除了线性层和激活函数以外，卷积神经网络还有一些特殊的层，如卷积层和池化层等。下面我们介绍一下这些特殊的层。

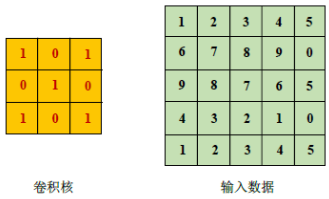
卷积层是卷积神经网络的特色，我们来看看其正向传播运算和反向传播如何进行。

前向传播

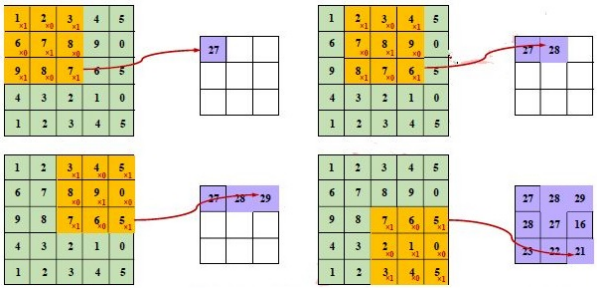
对于像我们本次实验完成的手写数字识别这种任务，其输入数据是手写数字的图片文件。因此，其输入数据是二维的。在计算机中，灰度图是一个$H \times W$的二维数据，其中每一个数据(像素点)是一个0到255之间的整数，表示其黑色深度；若是彩色图像，则是分为RGB(红绿蓝，即三原色)三组灰度图存储，表示其三原色各自的深度。

除了输入数据以外要执行卷积这一操作，我们还需要一个“卷积核”，卷积核的本质是一个大小为$K \times K$的矩阵。

我们先从灰度图(单张$H \times W$图片)开始讲起，下面是一个5×5输入数据和3×3卷积核的示意图:



卷积的操作即是让卷积核从左上角开始，每次“贴”在输入数据的一个区域上，然后让卷积核和输入数据对应位置的数字相乘并求和，作为卷积结果中一处的值。当卷积核在输入数据上滑动并遍历整个输入数据时，反复执行这一操作，就得到卷积的结果。其示意图如下所示:



这就是卷积的前向传播过程，卷积的结果也被称为**卷积特征**。

我们可以看到，对于$K \times K$大小的卷积核，卷积特征中的每一位只与输入数据中K^2个数据有关，而与其他数据均无关，这被称为卷积的**感受野**。这与我们的直觉是相符的——图片上某一个像素点与它周围的像素点关系较大，而与离他较远的像素点基本无关。

我们用符号$\otimes$表示卷积运算，则卷积过程可以表示为

$$y = x \otimes W$$

这里的W表示卷积核。

同时，与线性运算类似地，我们还可以给卷积特征图加上一个偏置(bias)，即给卷积特征图的每一位加上一个常数b，此时卷积过程可表示为

$$y = x \otimes W + b$$

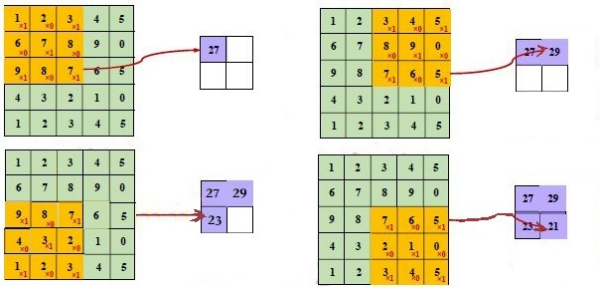
其中，卷积核W和偏置b都不是固定的，而是神经网络中同样可以尤梯度下降法进行优化的参数

步长(stride)

有时候，我们希望卷积之后的结果尺寸能迅速减小。一个简单的方法是，让卷积核在输入数据上滑动时，一下“跨过好几步”，而不是一步一步地滑动。而描述“每次跨过几步”的量就是步长。

如在上面的例子中，卷积核沿输入数据一步一步地滑动，就是步长stride=1.

下面是步长stride=2时的卷积特征示意图



未特别注明时，默认stride=1

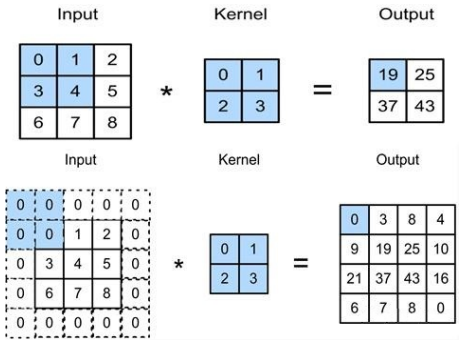
填充(padding)

有时，我们又希望卷积特征的尺寸不要减小得太快，或者希望卷积特征在尺寸上和原图保持一致之类。这时，我们就需要用到填充(padding)

填充的含义很简单，就是在原输入数据的基础上，在其上下左右四周加一圈或几圈数字，使得输入图像尺寸扩大，那么卷积结果的尺寸自然就扩大了。

在没有特殊说明时，padding操作使用的都是“0填充”，即在原图像的基础上在其上下左右加一圈或几圈0.

下面是一个padding=0和1的卷积对比图(其中stride=1):

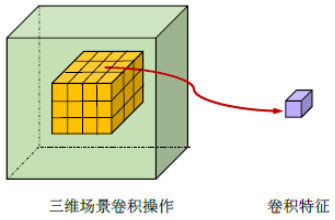


未特别注明时，默认padding=0(即不进行填充操作)

三维卷积

有时候，输入图像具有多个通道(Channels)，如RGB彩色图像就具有R,G,B三个通道，这时候就需要三维的卷积核。

三维图像的卷积仍是卷积核和输入数据对应位置的数字相乘求和，和二维卷积没有本质区别



假设输入数据有C个通道，则输入数据的尺寸为$C \times H \times W$，此时我们会选择使用$C \times K \times K$的卷积核，这样得到的结果就是$1 \times H' \times W'$的图像。

如果我们希望卷积特征仍有多个通道(是多维的)，我们可以使用多个不同的卷积核进行卷积操作。若使用了O个卷积核，就会得到O张$1 \times H' \times W'$的卷积特征图，此时卷积结果就是$O \times H' \times W'$的。

卷积结果的尺寸

我们已经知道，输入$C \times H \times W$的图像，经卷积过程可以得到$O \times H' \times W'$的卷积特征图。

而我们先前提到过，在训练神经网络时，在一个批次(batch)内可能同时输入多组数据进行训练。假如一个batch内输入N张图片，那么卷积输入图像的尺寸就会变成$N \times C \times H \times W$，相应地，卷积特征图会变成$N \times O \times H' \times W'$

设卷积过程使用的卷积核尺寸为$C \times K \times K$，步长stride=S，填充Padding=P，那么我们可以计算出，卷积特征图像尺寸中H', W'的值分别为:

$$H' = \frac{H - K + 2P}{S} + 1$$

$$W' = \frac{W - K + 2P}{S} + 1$$

反向传播

卷积看上去虽然有些复杂，但仔细一想，其实卷积仍然是一个线性的运算，这给我们的计算带来了方便。

为了方便，我们用单张、单通道二维图像的情况进行推导(C通道的情况是同理且结果一致的)，损失函数对卷积核中p行q列元素$W_{p,q}$的偏导数。根据导数的链式法则，我们有

$$\frac{\partial J}{\partial W_{p,q}} = \frac{\partial J}{\partial y_{11}}\frac{\partial y_{11}}{\partial W_{p,q}} + \frac{\partial J}{\partial y_{12}}\frac{\partial y_{12}}{\partial W_{p,q}} + \ldots + \frac{\partial J}{\partial y_{H,W}}\frac{\partial y_{H,W}}{\partial W_{p,q}} = \sum_{i,j}\frac{\partial J}{\partial y_{i,j}}\frac{\partial y_{i,j}}{\partial W_{p,q}}$$

同理有

$$\frac{\partial J}{\partial x_{p,q}} = \sum_{i,j}\frac{\partial J}{\partial y_{i,j}}\frac{\partial y_{i,j}}{\partial x_{p,q}}$$

$$\frac{\partial J}{\partial b_{p,q}} = \sum_{i,j} \frac{\partial J}{\partial y_{i,j}} \frac{\partial y_{i,j}}{\partial b_{p,q}}$$

对卷积核W而言，卷积特征的每一位都和它有关。我们先考虑W_{11}，考虑$y = x \otimes W + b$，回忆一下卷积的计算过程，我们知道y_{11}的计算式中含有$W_{11}x_{11}$项(其中S是步长stride)，y_{12}的计算式中含有$W_{11}x_{1,1+S}$项，y_{13}中含有$W_{11}x_{1,1+2S}$项，...

以此类推，我们知道$y_{i,j}$中含有$W_{11}x_{1+(i-1)S,1+(j-1)S}$项，于是有

$$\frac{\partial y_{i,j}}{\partial W_{11}} = x_{1+(i-1)S,1+(j-1)S}$$

从而

$$\frac{\partial J}{\partial W_{11}} = \sum_{i,j} \frac{\partial J}{\partial y_{i,j}} x_{1+(i-1)S,1+(j-1)S}$$

同理的，我们将其推广到卷积核W中任一元素$W_{p,q}$的情况，可以得到

$$\frac{\partial J}{\partial W_{p,q}} = \sum_{i,j} \frac{\partial J}{\partial y_{i,j}} x_{p+(i-1)S,q+(j-1)S}$$

我们仔细观察这个式子，你会发现，这个求和公式其实就是$\frac{\partial J}{\partial y}$这一偏导矩阵在输入图像数据$x$矩阵上以步长S滑动，进行卷积的结果。

这就说明，损失函数对卷积核的偏导事实上是以卷积特征为卷积核，对输入图像进行卷积的结果，用公式表示即

$$\frac{\partial J}{\partial W} = y \otimes x$$

损失函数对偏置的导数$\frac{\partial J}{\partial b}$则相对好求。因为卷积特征的每个元素都加上了一个$b$，所以这个导数事实上就是

$$\frac{\partial J}{\partial b} = \sum_{i,j} \frac{\partial J}{\partial y_{i,j}}$$

(如果是输入图片是C通道的，那么就是$\frac{\partial J}{\partial b} = \sum_{i,j,k} \frac{\partial J}{\partial y_{i,j,k}}$)

最后我们再来考虑损失函数对输入图像x的导数。卷积特征中每个元素只与其感受野内的x数据有关，而与其感受野外的x数据无关。因此，卷积特征中的每个元素与输入数据x中的K^2个元素是有关的(K是卷积核的边长)。

如果输入数据中的元素$x_{p,q}$在卷积特征中的元素$y_{i,j}$的感受野内，那么$y_{i,j}$的表达式就会相应地含有$W_{m,n}x_{p,q}$项，于是有

$$\frac{\partial y_{i,j}}{\partial x_{p,q}} = W_{m,n}$$

则我们可以写出

$$\frac{\partial J}{\partial x_{p,q}} = \sum_{m,n \in A} W_{m,n} \qquad A = \{m,n | \text{其中} x_{p,q} \text{在} y_{i,j} \text{的感受野内，且对应的项为} W_{m,n}x_{p,q}\}$$

这一表达式看上去很复杂，甚至没有一个统一的形式。但由于我们是从偏导矩阵$\frac{\partial J}{\partial y}$来计算$\frac{\partial J}{\partial x}$，编程时我们只需要先把偏导矩阵$\Delta PDJx$的元素全部置零，再依次遍历$y_{i,j}$，将其感受野内的$x_{p,q}$的偏导加上对应的$W_{m,n}$即可。所以实际上这个表达式在编程时是容易实现的。

我们总结一下，损失函数对W, b, x的导数分别是

$$\frac{\partial J}{\partial W} = y \otimes x$$

$$\frac{\partial J}{\partial b} = \sum_{i,j} \frac{\partial J}{\partial y_{i,j}}$$

$$\frac{\partial J}{\partial x_{p,q}} = \sum_{m,n \in A} W_{m,n} \qquad A = \{m, n | \text{其中} x_{p,q} \text{在} y_{i,j} \text{的感受野内，且对应的项为} W_{m,n} x_{p,q}\}$$

若输入图像含有C个通道，则$\frac{\partial J}{\partial b}$相应变为

$$\frac{\partial J}{\partial b} = \sum_{i,j,k} \frac{\partial J}{\partial y_{i,j,k}}$$

求出这些后，我们即可用$\frac{\partial J}{\partial W}$和$\frac{\partial J}{\partial b}$更新参数优化神经网络，并将$\frac{\partial J}{\partial x}$传回上一层继续导数的反向传播。

需要注意的是，以上公式中的x是padding过后的x数据。我们需要先求出这一导数矩阵，然后舍去padding的部分，仅保留原图像部分的导数作为$\frac{\partial J}{\partial x}$的实际值。

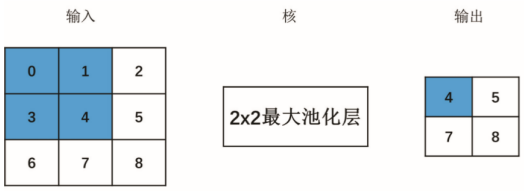
3.9 池化层

与卷积非常类似，池化也有“核”、步长stride、填充padding的概念。而与卷积操作不同的地方在于池化的“核”并不是靠对应位置的数相乘求和来得到结果。

池化的类型有多种，最常见的是最大池化(Max Pooling)和平均池化(Average Pooling)，我们的实验中也用到了这两种池化。

最大池化

最大池化就是取核覆盖到的所有元素中(即感受野中)最大的一个元素作为结果。下图是最大池化的一个示意图



最大池化的反向传播非常简单，由于$y_{i,j}$只与其感受野中最大的元素有关，而与其他元素无关，所以有

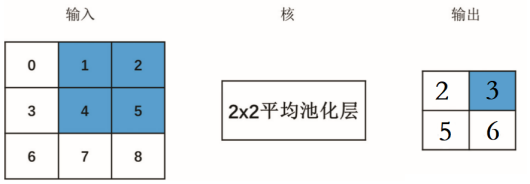
$$\frac{\partial y_{i,j}}{\partial x_{p,q}} = \begin{cases} 1 & x_{p,q} \text{是} y_{i,j} \text{感受野中的最大元素} \\ 0 & x_{p,q} \text{不是} y_{i,j} \text{感受野中的最大元素} \end{cases}$$

因此损失函数对x中各元素的导数为

$$\frac{\partial J}{\partial x_{p,q}} = \sum_{i,j \in A} \frac{\partial J}{\partial y_{i,j}} \qquad A = \{i, j | \text{其中} x_{p,q} \text{是} y_{i,j} \text{感受野中的最大元素}\}$$

平均池化

平均池化是取核覆盖到的所有元素(即感受野)的平均值作为结果。下图是平均池化的一个示意图



因为平均池化时，感受野内每个元素的权重相同，所以导数会均匀分配到感受野中的每一个元素上，而感受野内有$\frac{1}{K^2}$个元素，所以每个元素分的导数的$\frac{1}{K^2}$ (K是核的边长)。即平均池化的反向传播公式为

$$\frac{\partial J}{\partial x_{p,q}} = \frac{1}{K^2} \sum_{i,j \in A} \frac{\partial J}{\partial y_{i,j}} \qquad A = \{i, j | x_{p,q} \text{在} y_{i,j} \text{的感受野内}\}$$

平均池化的计算过程可以看作是以下面这个矩阵为卷积核的卷积过程

$$\frac{1}{K^2} \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{bmatrix}$$

事实上，在前向传播上，平均池化和以上述矩阵为卷积核的卷积完全一致。而它们的区别在于，平均池化里的核是固定的，不像卷积层里那样需要根据偏导更新优化参数。

3.10 输出层

在本次实验中，我们所处理的问题是一个分类问题，我们神经网络输出的结果是一个10维的向量p，其中各个元素分别表示神经网络判断输入图像是数字0,1,2,...,9的概率。我们取概率值最大的那个数字作为神经网络的预测结果。

既然输出的是概率向量，那么就存在要求

$$\sum_{i=0}^{9} p_i = 1$$

当隐藏层[3]给我们一个10维的向量时，里面的数据不一定能满足和为1的条件，我们需要对其进行归一化使其和为1。

我们将隐藏层输出的10维向量记作向量x，神经网络中最常用的一个归一化函数是Softmax函数。

Softmax函数的前向传播

Softmax函数又称为归一化指数函数，其计算如下:

$$p_j = \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}$$

[3]隐藏层即神经网络中间运算的那些层，线性层、激活函数层、卷积层、池化层都属于隐藏层

其中n是类别的总数，p_j即是分类维第j类的概率。

在我们这个问题中，类别是0,1,2,...,9，所以Softmax相应地就是

$$p_j = \frac{e^{x_j}}{\sum_{i=0}^{9} e^{x_i}}$$

但是我们注意到，指数函数的增长是非常快的，如果x向量中含有一个较大的整数，在指数运算之后就会变得非常大，这容易使得在运算时出现溢出情况或者丢失精度。因此为了避免这些情况的出现，我们通常会先对x向量进行非正化处理。我们首先找出x向量中最大的一个元素，然后从x向量中的所有元素中都减去这个最大元素，这样x向量中的所有元素就都是非正数了，这样就避免了取指数后数值更大导致溢出或精度丢失的问题。

因此，如果应用了非正化处理，我们实际上所使用的Softmax函数就是

$$p_j = \frac{e^{x_j - \max x_k}}{\sum_{i=0}^{9} e^{x_i - \max x_k}}$$

Softmax函数的反向传播

简便起见，我们这里采用没有进行非正化处理的Softmax函数进行反向传播的推导。

首先，我们回忆一下交叉熵损失函数

$$J = -\frac{1}{N} \sum a_i \log p_i$$

其中，N是数据量，a_i是正确结果，p_i是神经网络的输出结果。

方便起见，我们考虑单个样本的情况，则

$$J = -\sum a_i \log p_i$$

由于正确结果是一个确定的分类，所以其概率向量中，正确分类的分量为1，其他分量均为0，于是，损失函数即为

$$J = -\log p_k$$

其中k是输入图像的正确分类

所以容易看出，损失函数对神经网络输出层输出结果(概率向量p)的导数是[4]

$$\frac{\partial J}{\partial p_i} = \begin{cases} -\frac{1}{p_i} & i = k \\ 0 & i \neq k \end{cases}$$

所以$\frac{\partial J}{\partial x}$按链式法则等于

$$\frac{\partial J}{\partial x} = \sum_{i=0}^{9} \frac{\partial J}{\partial p_i} \frac{\partial p_i}{\partial x} = -\frac{1}{p_k} \frac{\partial p_k}{\partial x}$$

因为$p_k = \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}}$，所以根据除法的求导法则我们又有

$$\frac{\partial p_k}{\partial x_k} = \frac{e^{x_k} \sum_{i=1}^{n} e^{x_i} - e^{x_k} \cdot e^{x_k}}{(\sum_{i=1}^{n} e^{x_i})^2} = \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}} \cdot \frac{\sum_{i=1}^{n} e^{x_i} - e^{x_k}}{\sum_{i=1}^{n} e^{x_k}} = p_k(1 - p_k)$$

[4]此处推导中log取自然对数ln，即底数取为e。如果取了不同的底数，实质上只差常数倍，没有本质区别

$$\text{对于}j \neq k\text{,有} \quad \frac{\partial p_k}{\partial x_j} = \frac{0 - e^{x_j} \cdot e^{x_k}}{(\sum_{i=1}^{n} e^{x_i})^2} = -\frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}} \cdot \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}} = -p_j p_k$$

于是求得输出层的反向传播公式为

$$\frac{\partial J}{\partial x_i} = \begin{cases} p_k - 1 & i = k \\ p_j & i \neq k \end{cases}$$

输出层的反向传播公式也可简写为

$$\frac{\partial J}{\partial x} = p - a$$

其中a是正确结果的概率向量(即正确分类项为1，其他项全为0)

至此，我们求出了损失函数对最后一层的导数，只需要按前面推导出的各层的反向传播公式逐层反向传播，即可求出损失函数对每个参量的导数，从而按梯度下降法更新优化参数，达到训练神经网络的目的。

3.11 超参数

在前面的知识整理过程中，你可能注意到了两个特殊的变量。我们这里再追加一个特殊的变量训练轮数(epochs)，这三个特殊的变量在神经网络中有着重要意义。

- 每个批次(batch)输入的样本量N(称作Batch Size)
- 学习率lr(Learning Rate)
- 训练轮数(epoch)

这三个变量是神经网络的**超参数**，它们的值决定了神经网络的训练效果。下面我们来逐个梳理这些超参数的作用。

Batch Size

Batch Size是神经网络训练时，每一个批次(batch)输入的样本量。前面谈及输入x的大小时提到过“有时会将N个数据作为一个批次(batch)一起输入”，这里的N实际上就是batch size。

若训练集共有T个样本，Batch Size为B，那么显然，训练集可以被分为$\lceil \frac{T}{B} \rceil$个批次输入神经网络进行训练。

如果Batch Size较大，那么只需要进行较少批次的计算就可以完全遍历整个训练集，从而可以在更短的时间内完成神经网络的训练。当然，更大的Batch Size也意味着每批次输入的数据量更大，需要消耗更多的内存来存储这些数据，这是一个“空间换时间”的行为。

此外，当Batch Size增大的时候，也就意味着一次训练将很多个样本纳入了考虑，那么这些样本的一些“共性”就会被放大，而样本的一些特性则相对来说在某种程度上被忽略了。因此，较大的Batch Size会导致神经网络的泛性下降，表现为准确率降低。

综上，较小的Batch Size可以提高神经网络模型的准确性，但是会延长训练时间。

所以Batch Size的调节是在泛用性/准确率和训练耗时之间作一个平衡。

Learning Rate

我们在更新参数时，是按下面的方式更新的

$$W = W - \frac{\partial J}{\partial W} * lr$$

其中lr即为模型的学习率(Learning Rate)

在训练前期，较大的学习率可以让模型参数快速靠近最小值，能较快地提高模型准确率；然而，学习率太大可能导致损失函数变化较大，或者导致模型训练后期在最小值附近打转，使模型难以收敛。

但是若学习率太小，可能导致模型陷入局部极小值而无法找到全局最小值，或者导致模型收敛速度很慢，需要较多的轮次才能收敛，使得训练耗时延长。

对于一个神经网络而言，学习率太大太小都不好，需要在实验中调节学习率使模型达到最佳效果。

此外，我们注意到模型训练前期需要学习率相对较大，但是后期需要学习率相对较小。所以我们也可以把学习率设置为一个变量，随着训练的进行而不断减小，以达到更优的训练效果。

Epochs

Epochs即训练轮次表示训练集的样本数据将会被输入神经网络用于训练多少次。

当然，每一轮(epoch)中，样本数据的顺序会被打乱，这意味着虽然训练集样本本身没有变化，但每一轮中哪些样本在同一批次(batch)中被输入模型是被打乱了的。

如果Epochs太小，模型可能还没有收敛到最佳效果；而如果Epochs太大，则模型训练耗时会很长，而且最后的一些轮次中模型效果可能没有明显提高。

因此，也有一个比较好的办法是不把Epochs设为定值，而是记录每一轮的损失函数值或训练准确度，当某一轮损失函数值的减少或训练准确度的提高小于某个阈值时，则停止训练。

4 代码实现

下面，我们根据前面整理的这些理论来实现神经网络程序的编写。

4.1 激活函数

激活函数的代码实现是较为简单的，只需要记住我们所使用的几种激活函数的表达式和导数公式即可，这里不再复述了。

```
class ReLU:
    def forward(self, x):
        res = x if x > 0 else 0
        self.x = x
        self.y = res
        return self.y

    def backward(self, dy):
        d = 1 if self.x > 0 else 0
        self.derivative = d * dy
```

```
        return self.derivative


class Tanh:
    def forward(self, x):
        res = math.tanh(x)
        self.x = x
        self.y = res
        return self.y

    def backward(self, dy):
        d = 1 - self.y * self.y
        self.derivative = d * dy
        return self.derivative


class Sigmoid:
    def forward(self, x):
        res = 1 / (1 + np.exp(-x))
        self.x = x
        self.y = res
        return self.y

    def backward(self, dy):
        d = self.y * (1 - self.y)
        self.derivative = d * dy
        return self.derivative
```

4.2 卷积层

对于卷积层，让我们回顾一下其前反向传播的公式。

其前向传播的公式为

$$y = x \otimes W + b$$

其导数和反向传播公式为

$$\frac{\partial J}{\partial W} = y \otimes x$$

$$\frac{\partial J}{\partial b} = \sum_{i,j} \frac{\partial J}{\partial y_{i,j}}$$

$$\frac{\partial J}{\partial x_{p,q}} = \sum_{m,n \in A} W_{m,n} \qquad A = \{m, n | \text{其中} x_{p,q} \text{在} y_{i,j} \text{的感受野内，且对应的项为} W_{m,n} x_{p,q}\}$$

在编程时有一个需要注意的地方:对于卷积运算，我们确实可以通过很多个for循环的嵌套来实现。然而，我们知道，Python是一个运行很慢的语言，所以如果堆叠了大量for循环嵌套，那么这个神经网络的训练和运行将会消耗大量的时间。(理论上来说，也可以用C++等运行更快的语言来实现神经网络。但因为Python在人工智能领域具有更好的生态，所以神经网络基本都还是基于Python实现的)

因此，为了缩短程序的运行时间，我们需要尽可能地少使用for循环。为了实现这一点，我们要尽可能地对x, W, b这些参变量进行**向量化**，并用numpy的向量化运算进行计算，这样就能大幅提高代码的运算速度。(这是因为numpy的底层主要是由C语言实现的，运算速率会比Python高很多)

还需要注意，卷积层进行反向传播时，要先求出对padding后图像的导数，再去除padding操作中填充的数，只保留对原图像中各元素的导数。

```
class Conv2d:
    def __init__(self, in_channels: int, out_channels: int, kernel_size: int,
                 stride: int = 1, padding: int = 0, dtype = None):
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding
        self.dtype = dtype
        self.W = np.random.randn(out_channels, in_channels, kernel_size, kernel_size).astype(dtype) #O*C*kernel_size*
            kernel_size
        self.b = np.random.randn(out_channels).astype(dtype)
        return

    def forward(self, x):
        """
        x - shape (N, C, H, W)
        return the result of Conv2d with shape (N, O, H', W')
        """
        N, C, H, W = x.shape
        O = self.out_channels
        padding = self.padding
        self.x = x
        #执行padding操作，在H,W方向前后填0
        self.pad_img = np.pad(x, [[0, 0], [0, 0], [padding, padding], [padding, padding]], 'constant')
        H_ = (H - self.kernel_size + 2 * self.padding) // self.stride + 1
        W_ = (W - self.kernel_size + 2 * self.padding) // self.stride + 1
        self.y = np.zeros([N, O, H_, W_], dtype = self.dtype)
        for h in range(H_):
            for w in range(W_):
                row_min = h * self.stride
                row_max = h * self.stride + self.kernel_size
                col_min = w * self.stride
                col_max = w * self.stride + self.kernel_size
                sub_fig = self.pad_img[:, :, row_min:row_max, col_min:col_max] #对应子图
                for j in range(O):
                    self.y[:, j, h, w] = np.sum(sub_fig*self.W[j, :, :, :], axis=(1,2,3)) + self.b[j]
        return self.y

    def backward(self, dy, lr):
        """
        dy - the gradient of last layer with shape (N, O, H', W')
        lr - learning rate
        calculate self.w_grad to update self.weight,
        calculate self.b_grad to update self.bias,
        return the result of gradient dx with shape (N, C, H, W)
        """
        N, O, H_, W_ = dy.shape
        N, C, H_pad, W_pad = self.pad_img.shape
        N, C, H, W = self.x.shape
        self.pad_dx = np.zeros([N, C, H_pad, W_pad], dtype = self.dtype) #含padding的梯度
```

```
        self.dW = np.zeros([O, C, self.kernel_size, self.kernel_size], dtype = self.dtype)
        self.db = np.zeros(O, dtype = self.dtype)
        for h in range(H_):
            for w in range(W_):
                row_min = h * self.stride
                row_max = h * self.stride + self.kernel_size
                col_min = w * self.stride
                col_max = w * self.stride + self.kernel_size
                sub_fig = self.pad_img[:, :, row_min:row_max, col_min:col_max] #对应子图
                for j in range(O):
                    self.dW[j, :, :, :] += np.sum(sub_fig * (dy[:, j, h, w])[:, None, None, None], axis = 0)
                    self.db[j] += np.sum(dy[:, j, h, w], axis = 0) #N维变1维
                    self.pad_dx[:, :, row_min:row_max, col_min:col_max] += self.W[j, :, :, :] * (dy[:, j, h, w])[:,
                        None, None, None]
                    #padding过后的梯度
        self.W -= lr * self.dW #更新W,b
        self.b -= lr * self.db
        self.dx = self.pad_dx[:, :, self.padding:self.padding+H, self.padding:self.padding+W] #原图梯度
        return self.dx
```

4.3 池化层

最大池化

最大池化的正向传播是取感受野中的最大值，而反向传播则是将$\frac{\partial J}{\partial y}$中各元素的导数全部分给其感受野中对应最大的那个元素。

在池化层中，仍然是为了加快运算速率，我们还是需要将其中用到的参变量进行向量化。方便的是，numpy中提供了np.max,np.avg等统计函数，可以方便地求出矩阵中的最大值或平均值，这对池化操作非常有帮助。

```
class MaxPool2d:
    def __init__(self, kernel_size: int, stride = None, padding = 0):
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding


    def forward(self, x):
        """
        x - shape (N, C, H, W)
        return the result of MaxPool2d with shape (N, C, H', W')
        """
        N, C, H, W = x.shape
        self.size = [N, C, H, W]
        H_ = (H - self.kernel_size + 2*self.padding) // self.stride + 1
        W_ = (W - self.kernel_size + 2*self.padding) // self.stride + 1
        padding = self.padding
        self.x = x
        self.pad_img = np.pad(x, [[0, 0], [0, 0], [padding, padding], [padding, padding]], 'constant')
        self.y = np.zeros([N, C, H_, W_])#按尺寸创建结果数组
        self.maxindex = np.zeros([N, C, H_, W_])#存放最大值对应的索引，反向传播时用
        for h in range(H_):
```

```
            for w in range(W_):
                row_min = h * self.stride
                row_max = h * self.stride + self.kernel_size
                col_min = w * self.stride
                col_max = w * self.stride + self.kernel_size
                sub_fig = self.pad_img[:, :, row_min:row_max, col_min:col_max]
                self.y[:, :, h, w] = np.max(sub_fig, axis = (2,3))
        return self.y

    def backward(self, dy):
        """
        dy - shape (N, C, H', W')
        return the result of gradient dx with shape (N, C, H, W)
        """
        N, C, H_, W_ = dy.shape
        N, C, H, W =self.x.shape
        N, C, H_pad, W_pad = self.pad_img.shape
        self.pad_der = np.zeros([N, C, H_pad, W_pad])
        for h in range(H_):
            for w in range(W_):
                row_min = h * self.stride
                row_max = h * self.stride + self.kernel_size
                col_min = w * self.stride
                col_max = w * self.stride + self.kernel_size
                sub_fig = self.pad_img[:, :, row_min:row_max, col_min:col_max]
                max_fig = np.max(sub_fig, axis = (2,3), keepdims = True) #子图最大值
                isMax = (max_fig == sub_fig) #判断该处是否是最大值
                self.pad_der[:, :, row_min:row_max, col_min:col_max] += isMax * (dy[:, :, h, w])[:, :, None, None]
        self.derivative = self.pad_der[:, :, self.padding:self.padding+H, self.padding:self.padding+W]
        return self.derivative
```

平均池化

平均池化的前向传播是取其感受野中数据的平均值作为池化结果;相应地，平均池化的反向传播是将$\frac{\partial J}{\partial y}$平均分配到其感受野中的各个元素上去。

```
class AvgPool2d:
    def __init__(self, kernel_size: int, stride = None, padding = 0):
        self.kernel_size = kernel_size
        self.stride = stride if stride else kernel_size
        self.padding = padding
        return

    def forward(self, x):
        """
        x - shape (N, C, H, W)
        return the result of AvgPool2d with shape (N, C, H', W')
        """
        N, C, H, W = x.shape
        self.size = [N, C, H, W]
        H_ = (H - self.kernel_size + 2 * self.padding) // self.stride + 1
        W_ = (W - self.kernel_size + 2 * self.padding) // self.stride + 1
        padding = self.padding
        self.x = x
```

```
        self.pad_img = np.pad(x, [[0, 0], [0, 0], [padding, padding], [padding, padding]], 'constant')
        self.y = np.zeros([N, C, H_, W_])
        for h in range(H_):
            for w in range(W_):
                row_min = h * self.stride
                row_max = h * self.stride + self.kernel_size
                col_min = w * self.stride
                col_max = w * self.stride + self.kernel_size
                sub_fig = self.pad_img[:, :, row_min:row_max, col_min:col_max]
                self.y[:, :, h, w] = np.mean(sub_fig, axis = (2,3))
        return self.y

    def backward(self, dy):
        """
        dy - shape (N, C, H', W')
        return the result of gradient dx with shape (N, C, H, W)
        """
        N, C, H_, W_ = dy.shape
        N, C, H, W = self.size
        N, C, H_pad, W_pad = self.pad_img.shape
        self.pad_der = np.zeros([N, C, H_pad, W_pad]) #padding后图像的梯度
        for h in range(H_):
            for w in range(W_):
                row_min = h * self.stride
                row_max = h * self.stride + self.kernel_size
                col_min = w * self.stride
                col_max = w * self.stride + self.kernel_size
                for row in range(row_min, row_max):
                    for col in range(col_min, col_max):
                        self.pad_der[:, :, row, col] += dy[:, :, h, w] / (self.kernel_size * self.kernel_size)
        self.derivative = self.pad_der[:, :, self.padding:self.padding+H, self.padding:self.padding+W]
        return self.derivative
```

4.4 线性层

线性层的前向传播公式为

$$y = xW + b$$

导数和反向传播公式为

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial J}{\partial y}W^T$$

$$\frac{\partial J}{\partial W} = x^T\frac{\partial J}{\partial y}$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} = \frac{\partial J}{\partial y}$$

```
class Linear:
    def __init__(self, in_features: int, out_features: int, bias: bool = True):
        self.in_features = in_features
        self.out_features = out_features
        self.bias = bias
```

```
        self.W = np.random.randn(in_features, out_features).astype(np.float32)
        self.b = np.random.randn(out_features).astype(np.float32)
        return

    def forward(self, x):
        """
        x - shape (N, C)
        return the result of Linear layer with shape (N, O)
        """
        self.x = x
        self.y = np.dot(x, self.W)
        if self.bias:
            self.y += self.b
        return self.y

    def backward(self, dy, lr):
        """
        dy - shape (N, O)
        return the result of gradient dx with shape (N, C)
        """
        N, O = dy.shape
        self.dW = np.dot(self.x.T, dy) #梯度计算
        self.db = np.sum(dy, axis = 0) #变为1*O的导数
        self.dx = np.dot(dy, self.W.T)
        self.W = self.W - lr * self.dW
        if self.bias == True: #需要偏置时才更新b
            self.b = self.b - lr * self.db
        return self.dx
```

4.5 输出层

我们先前已经推导过，在我们的数字分类问题中，Softmax函数计算式为

$$p_j = \frac{e^{x_j}}{\sum_{i=0}^{9} e^{x_i}}$$

交叉熵损失函数的计算式为

$$J = -\frac{1}{N}\sum \log p_k$$

其中p_k是神经网络输出的对正确类别的概率。

而其对应的反向传播公式为

$$\frac{\partial J}{\partial x} = p - a$$

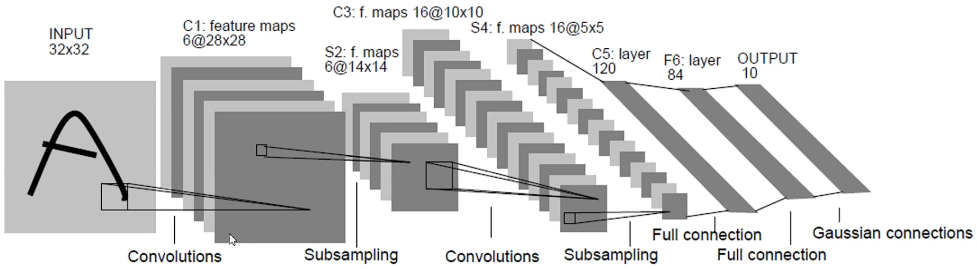
```
class CrossEntropyLoss:
    def __call__(self, x, label):
        n = label.shape[0] #样本个数
        p = np.exp(x)
        p = p / np.sum(p, axis = 1, keepdims = True) #Softmax
        LOSS = 0
        for i in range(n):
            LOSS -= math.log(p[i, label[i]])
        LOSS = LOSS / n

```

```
        dx = p.copy()
        for i in range(n):
            dx[i, label[i]] -= 1
        return LOSS, dx
```

4.6 网络构建

LeNet5卷积神经网络的结构如下图所示



我们按顺序逐层搭建和连接这一神经网络结构即可(注意由于数据矩阵尺寸的问题，中间需要对数据进行一次reshape操作)

当然，除了LeNet结构以外，我们也可以按其他顺序和层数来连接和搭建神经网络中的这些“层”来探索更好的模型效果...当然，这就是本次实验之后的后话了。

```
class LeNet5:
    def __init__(self):

        self.conv1 = Conv2d(1, 6, 5, 1, 2)
        self.relu1 = Sigmoid()
        self.pool1 = AvgPool2d(2)
        self.conv2 = Conv2d(6, 16, 5)
        self.relu2 = Sigmoid()
        self.pool2 = AvgPool2d(2)
        self.fc1 = Linear(16*5*5, 120)
        self.relu3 = Sigmoid()
        self.fc2 = Linear(120, 84)
        self.relu4 = Sigmoid()
        self.fc3 = Linear(84, 10)

    def forward(self, x):
        x = self.conv1.forward(x)
        x = self.relu1.forward(x)
        x = self.pool1.forward(x)
        x = self.conv2.forward(x)
        x = self.relu2.forward(x)
        x = self.pool2.forward(x)
        x = x.reshape(x.shape[0], -1)
        x = self.fc1.forward(x)
        x = self.relu3.forward(x)
        x = self.fc2.forward(x)
        x = self.relu4.forward(x)
        x = self.fc3.forward(x)
        return x
```

```
30
31      def backward(self, dy, lr):
32          dy = self.fc3.backward(dy, lr)
33          dy = self.relu4.backward(dy)
34          dy = self.fc2.backward(dy, lr)
35          dy = self.relu3.backward(dy)
36          dy = self.fc1.backward(dy, lr)
37          dy = dy.reshape(-1, 16, 5, 5)
38          dy = self.pool2.backward(dy)
39          dy = self.relu2.backward(dy)
40          dy = self.conv2.backward(dy, lr)
41          dy = self.pool1.backward(dy)
42          dy = self.relu1.backward(dy)
43          dy = self.conv1.backward(dy, lr)
```

4.7 主程序

主程序需要完成的操作有:

- 加载MNIST数据集图像数据
- 设置超参数LR,BS,Epochs
- 使用训练集进行训练(每一epoch中要记得随机重排训练集数据顺序)
- 训练好后，使用测试集测试准确率

```
1   def load_mnist(path, kind='train'):
2       image_path = glob.glob('./%s*3-ubyte' % (kind))[0]
3       label_path = glob.glob('./%s*1-ubyte' % (kind))[0]
4
5       with open(label_path, "rb") as lbpath:
6           magic, n = struct.unpack('>II', lbpath.read(8))
7           labels = np.fromfile(lbpath, dtype=np.uint8)
8
9       with open(image_path, "rb") as impath:
10          magic, num, rows, cols = struct.unpack('>IIII', impath.read(16))
11          images = np.fromfile(impath, dtype=np.uint8).reshape(len(labels), 28*28)
12
13      return images, labels
14
15  if __name__ == '__main__':
16      train_images, train_labels = load_mnist("mnist_dataset", kind="train")
17      test_images, test_labels = load_mnist("mnist_dataset", kind="t10k")
18      train_images = train_images.astype(np.float16) / 256
19      test_images = test_images.astype(np.float16) / 256
20
21      cnn = LeNet5()
22      LossFunction = CrossEntropyLoss()
23      lr = 0.0015 #超参数
24      epochs = 10
25      batch_size = 64
26      train_images = train_images.reshape(-1, 1, 28, 28) #导入图片
```

```
    test_images = test_images.reshape(-1, 1, 28, 28)
    Accuracy = np.zeros(epochs)
    Loss = np.zeros(epochs)
    print(f"Len train_images:{len(train_images)}, batch_size = {batch_size}")

    for epoch in range(epochs):
        order = np.random.permutation(len(train_images)) #随机顺序
        train_images = train_images[order]
        train_labels = train_labels[order]
        epoch_loss = 0
        correct = 0 #这一epoch中的正确数

        print(f"Now start training epoch {epoch+1}/{epochs}")
        i_range = range(0, len(train_images), batch_size)
        i_range = tqdm.tqdm(i_range) #进度条
        for i in i_range:
            images_batch = train_images[i:i+batch_size] #这一batch的图片和标签
            labels_batch = train_labels[i:i+batch_size]

            #前向传播
            outputs = cnn.forward(images_batch)
            loss, dy = LossFunction(outputs, labels_batch)
            epoch_loss += loss

            #反向传播
            cnn.backward(dy, lr)

            #统计这一batch的预测正确性
            predictions = np.argmax(outputs, axis=1)
            correct += np.sum(predictions == labels_batch)

        accuracy = correct / len(train_images) #这一epoch的预测准确率
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {epoch_loss:.4f}, Accuracy: {100 * accuracy:.4f}%")
        Accuracy[epoch] = accuracy
        Loss[epoch] = epoch_loss
    x = np.arange(epochs)
    plt.plot(x, Accuracy) #绘制epoch准确率曲线
    plt.savefig('Accuracy.png', dpi=300)

    #测试集检验
    correct = 0
    total = 0
    for i in range(0, len(test_images), batch_size):
        x_batch = test_images[i:i + batch_size]
        y_batch = test_labels[i:i + batch_size]

        outputs = cnn.forward(x_batch)
        predictions = np.argmax(outputs, axis=1)
        correct += np.sum(predictions == y_batch)
        total += len(y_batch)
    print(f'current setting: epochs={epochs}, batch-size={batch_size}, learning-rate={lr}')
    print(f'{total} test data, {correct} correct')
    print(f'Test Accuracy: {100 * correct / total:.4f}%')
```

5 实验结果和超参数调节分析

5.1 实验结果

由于我的电脑是轻薄本，性能较为一般，且无法用*GPU*训练，只能使用*CPU*运行和训练神经网络，所以速度较慢。因此，出于训练耗时考虑，只能选择较小的*Epochs*，还请见谅。

在本次实验中，我尝试使用了多种超参数不同取值的组合来对卷积神经网络进行训练，文件夹的“Accuracy Curve”文件夹中有不同超参数的训练集准确率随epoch的变化曲线。

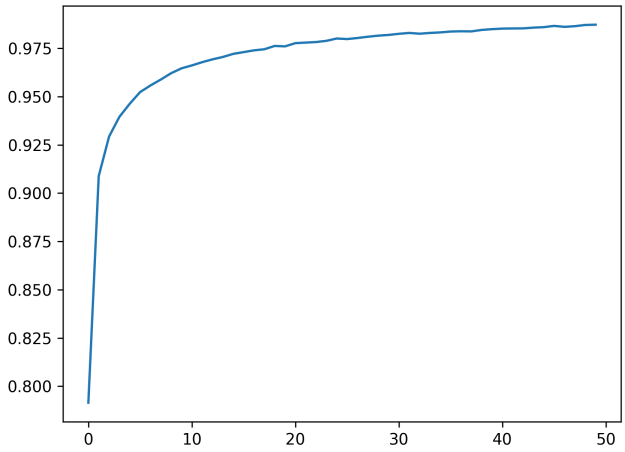
其中，准确率最高的一次所采用的超参数为

Learning Rate = 0.0015

Batch Size = 64

Epochs = 50

该次训练最终得到的**测试集分类准确率为98.17%**，其训练集准确率随epoch的变化曲线如下



从图中可以看出，此时的准确率仍有一定的上升空间，若能将Epochs提升至100~200，准确率有望提升至99%

5.2 超参数调节分析

为研究各超参数的变化对神经网络训练效果的印象，我们采用控制变量法，保持学习率、Batch Size、Epochs中的两个不变，观察另一个超参数取不同值时神经网络的情况。各个超参数的分析如下。

学习率

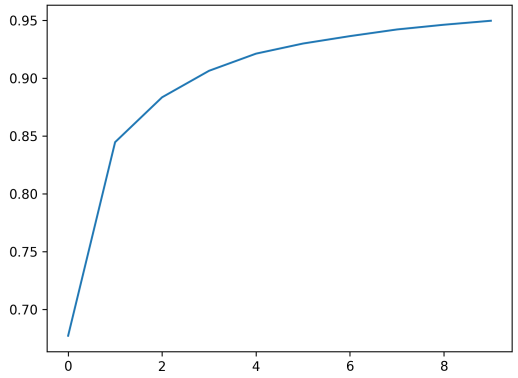
我们控制Batch Size=32， Epochs=10保持不变

将学习率Learning Rate分别设置为0.001,0.0015,0.002,0.005，观察并分析神经网络的训练效果

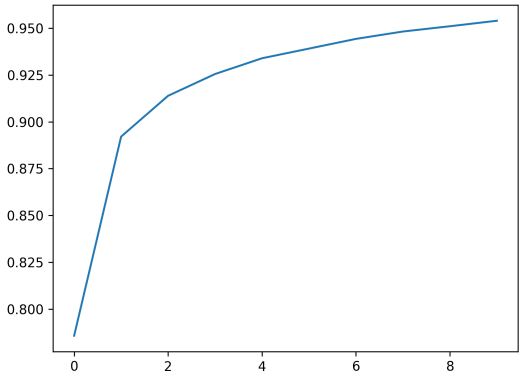
它们在测试集上的预测准确率情况如下:

Batch Size	Test Accuracy
0.001	95.58%
0.0015	95.83%
0.002	96.15%
0.005	96.93%

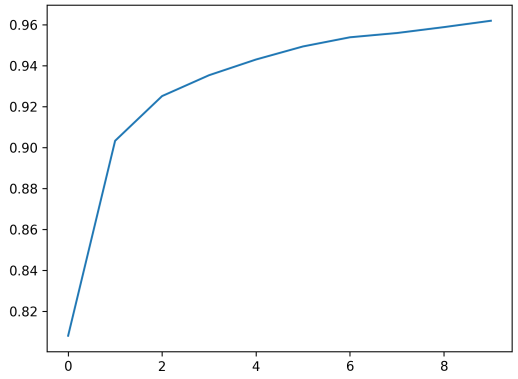
下面是各学习率在训练集中的分类准确率随epoch的变化情况:



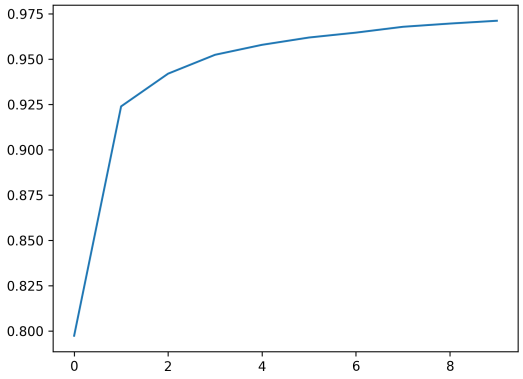
Learning Rate = 0.001



Learning Rate = 0.0015



Learning Rate = 0.002



Learning Rate = 0.005

在上述结果中，我们发现测试集准确率随着学习率的增大而呈现上升趋势。我们需要注意的是，**Epochs=10是一个较少的轮次，所以模型还未完全收敛。**所以上述几种情况的差异主要在于，学习率大的训练效果收敛更充分，而学习率小的收敛不充分。

通过上述对比可以看出，学习率主要影响模型的收敛速度。学习率较大时，则可以在更少的轮次中快速收敛

简而言之，**(在一定范围内)学习率越高，模型收敛越快。**

但学习率如果太高，也会带来问题，即在模型训练后期容易在最小值附近徘徊，而不能真正达到最小值(这是因为学习率太大时，梯度下降中每次移动的步长太长，容易“跨过”最小值)。此外，学习率太高还容易导致损失函数震荡过快，引起梯度爆炸问题。

学习率太小除了收敛慢以外，也容易导致模型迅速落入局部极小值，难以找到全局最小值。

所以**对于模型训练的前期，学习率大一些好，此时可以使模型较快地收敛并有利于避开局部极小值；而在训练后期，则是学习率小一些好，这样可以使模型训练的最终结果更加靠近最小值。**

Batch Size

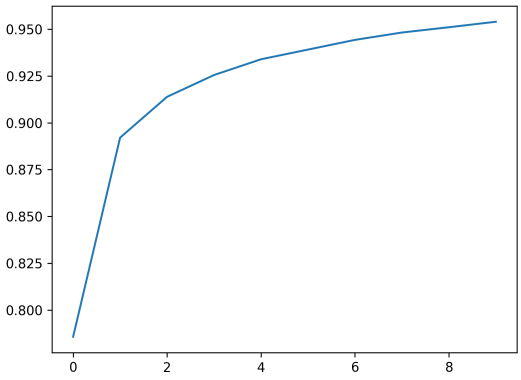
我们控制学习率lr=0.0015，Epochs=10保持不变

将Batch Size分别设置为32,64,128,256，观察并分析神经网络的训练效果

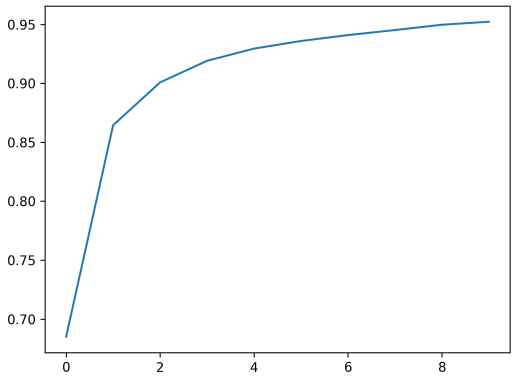
它们在测试集上的预测准确率和每批次耗时情况分别为[5]

Batch Size	Accuracy	Time Per Epoch
32	95.83%	13min
64	95.72%	9min
128	95.45%	6min
256	94.84%	5min

下面是各Batch Size在训练集中预测准确率随批次(epoch)的变化情况

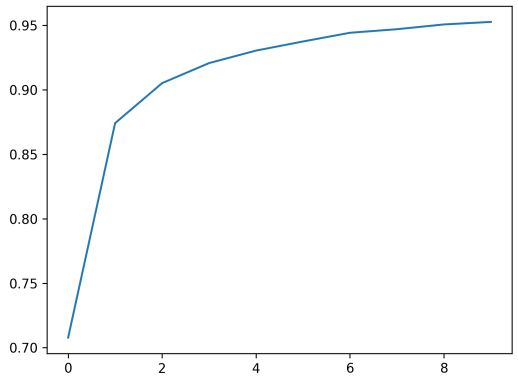


Batch Size = 32

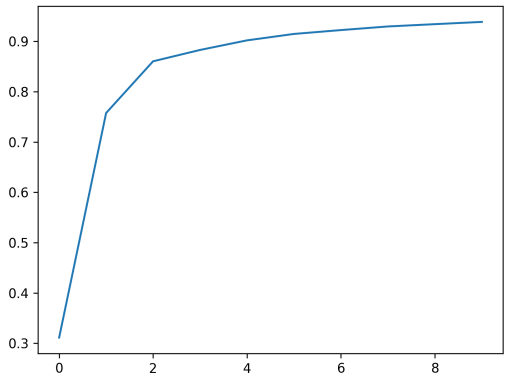


Batch Size = 64

[5]耗时情况是在我本机使用CPU运行时的平均耗时情况，对于不同性能的计算机这个时间可能有所不同，但耗时趋势应当是一致的。



Batch Size = 128



Batch Size = 256

从上述对比中可以看出，**Epochs越大，模型训练耗时越短，但准确率也越低，占用内存也越大。**

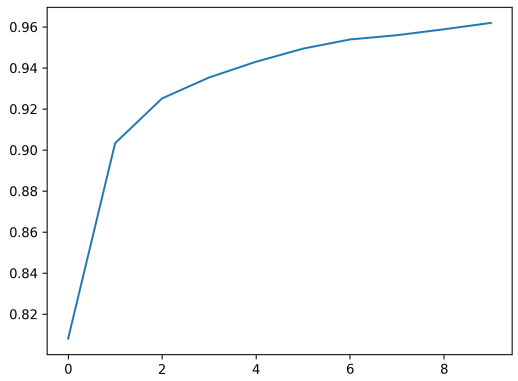
这是因为如果Batch Size约大，就可以在较少的batch内完成一个epoch，从而耗时就相对缩短了。而更大的Batch Size也意味着每批次输入的数据量更大，需要消耗更多的内存来存储这些数据，这是一个“空间换时间”的行为。

此外，当Batch Size增大的时候，也就意味着一次训练将很多个样本纳入了考虑，那么这些样本的一些“共性”就会被放大，而样本的一些特性则相对来说在某种程度上被忽略了。因此，较大的Batch Size会导致神经网络的泛性下降，表现为准确率降低。

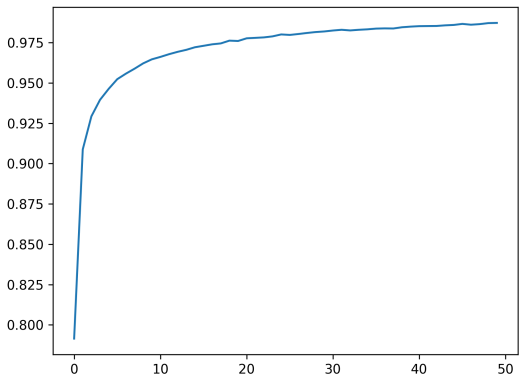
所以对Batch Size值的选择主要就是在模型准确率和训练速度上作出平衡(准确率达标的前提下，可以让BS值尽可能大以缩短训练时间；在训练时间可以接受的前提下，可以让BS值尽可能地小以提高模型准确率)

Epochs

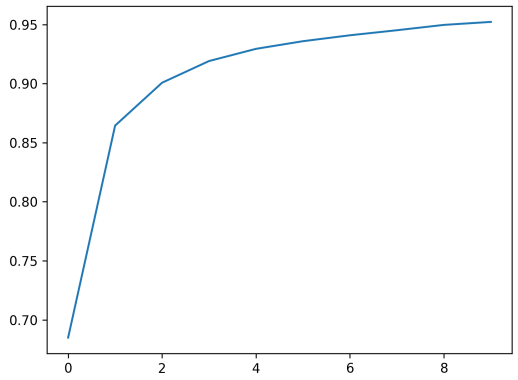
我们对于Learning Rate=0.002,Batch Size=32和Learning Rate=0.015,Batch Size=64两种情况，分别取Epochs为10和50对比效果，结果如下:



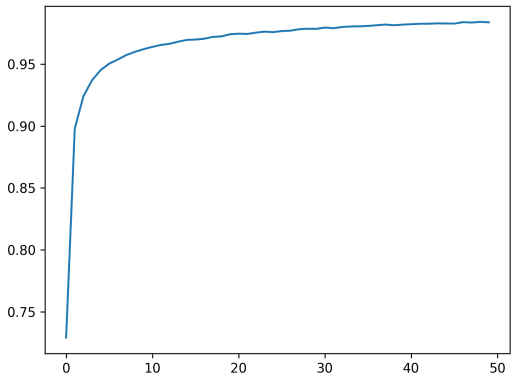
LR=0.002,BS=32,Epochs=10



LR=0.002,BS=32,Epochs=50



LR=0.0015,BS=64,Epochs=10



LR=0.0015,BS=64,Epochs=50

Condition	Epochs	Accuracy
LR=0.002,BS=32	10	96.15%
	50	98.13%
LR=0.0015,BS=64	10	95.72%
	50	98.17%

可以看出，在其他参数相同时，增大Epochs的值，准确率有所上升。出现这一结果的原因是**模型还未完全收敛**。

当模型最终收敛到最小值附近时，增大Epochs几乎不能再提高准确率(从上面的曲线也可以看出，随着epoch的增大，准确率的提升是越来越小的)。而模型的收敛速度(需要多少轮才收敛)，则主要由学习率决定了。

显然，Epochs的值不能太小，否则模型还未完成收敛，准确率会较低；但若Epochs太大，则后面的轮次中模型分类准确率增幅很小甚至没有提升，导致白白浪费训练时间。

因此，需要选择恰当的Epochs值，使模型在能够收敛到最小值附近的前提下轮次尽可能少。

6 进一步优化的方法

由于时间关系，我还有一些进一步优化和提高模型准确率的想法未能有时间在编程实践上实现，所以写在这里作为分析和参考，读者可以修改代码，自行尝试其效果。

动态学习率

我们留意到，在模型训练前期，我们会希望学习率稍微大一些，以使得模型快速收敛并防止落入局部最小值；而在模型训练后期，模型已经到最小值附近，此时我们则希望学习率稍微小一些，以对模型参数进行“微调”，使其更加靠近最小值。

简而言之，我们希望学习率在训练前期较大，在训练后期较小，以达到更好的训练效果。

所以，我们可以将学习率设置为一个动态的变量，并随epoch的增加而减小。

这样，理论上而言我们就可以获得更优的模型效果。

需要进一步探索的因素:学习率和epoch之间建立怎样的函数关系能使模型效果最好?

动态Epochs

类似地，我们也可以将Epochs由预先设定的常数转为一个动态量。

我们知道，Epochs太小会导致模型收敛不充分，而太大则会出现后面的轮次模型效果提高不明显，浪费训练时间的情况。

对此，我们可以不预先设定Epochs数，而是设定某一个迭代终止阈值。我们在训练模型时记录下每一epoch的准确率(或损失函数值)，当两轮之间准确率的提高量(或损失函数的减少量)小于这个阈值时，则停止训练。

这样的方法能够在保证模型训练充分度达到要求阈值的前提下尽可能地减少模型训练消耗的时间。

改用Leaky ReLU函数

在我们本次实验建立的神经网络中，我们使用了ReLU函数。而我们知道，ReLU函数存在有时会使神经元死亡并引起梯度消失的缺点，这可能会对模型的效果有一定的影响。

因此，我们可以考虑使用Leaky ReLU函数代替神经网络中的ReLU函数。它理论上具有ReLU的所有优点，并可以缓解神经元死亡和梯度消失的问题，可能可以在模型中带来更优的效果。