机器学习大作业——深度学习

组长：杨航（1811451） 组员：阮志涵（1811415）

初级要求

一、反向传播推导

因为我们的反向传播相当于是个优化问题，优化问题自然可以用经典的梯度下降法进行计算。

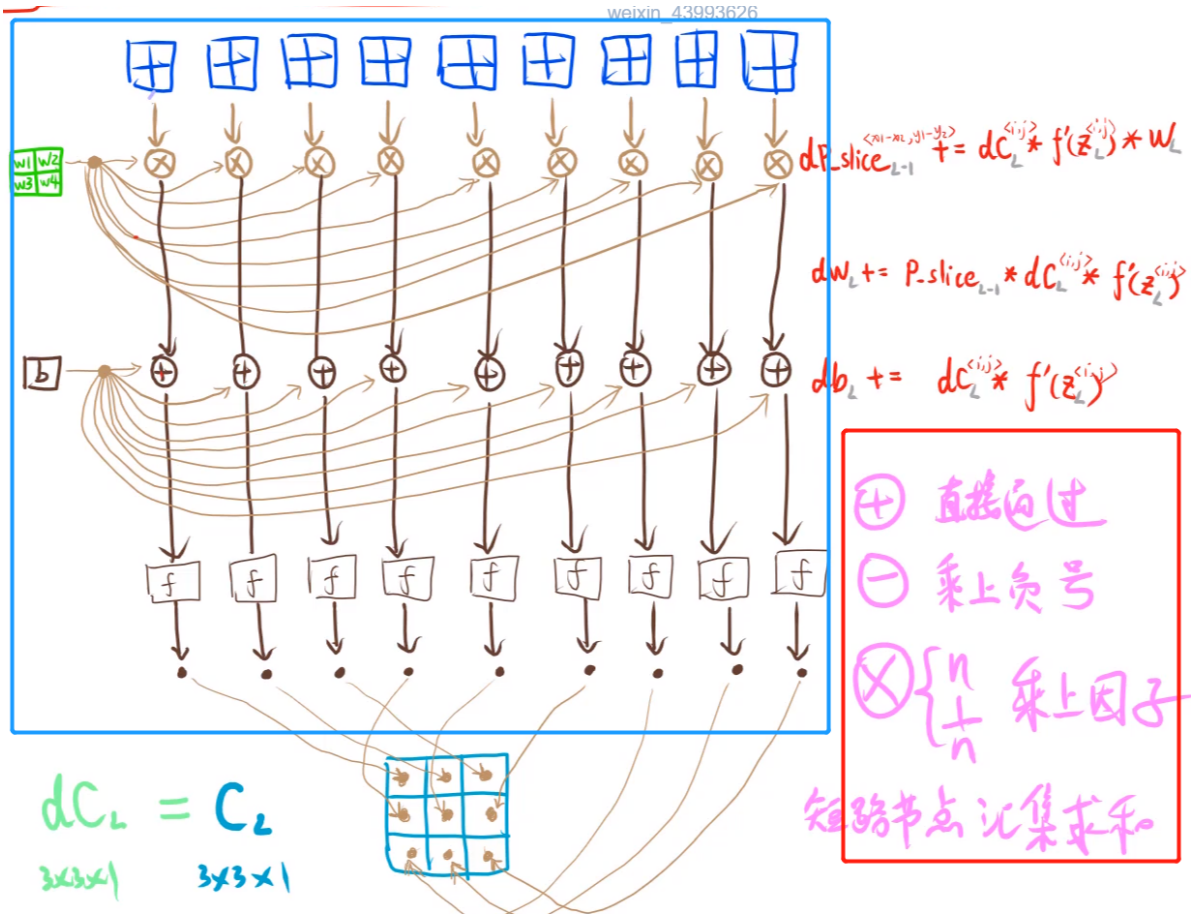
而梯度下降最常用的一种方法就是：链式求导法则，这样就能让梯度一层一层地传播下去。

而我们在神经网络中，要进行优化的主要是两个参数：w，b。

而我们的链式求导法则有几个非常有趣的规律：

1. 遇见 + 就直接传递（相当于短路）
2. 遇见 × 就用上一层的梯度乘上与需要求导的变量的系数
3. 遇见 - 就乘上-1，

   例如下图是卷积神经网络的计算图（蓝色框），以及用上述法则推导出的梯度（侧边红字），以及右下角是上述三个规则（红色框）。



1. FC层反向传播

FC层就是一个点乘操作和一个加法操作

- w遇见乘法，那dw=输入的特征a乘上上一层传下来的梯度，
- b遇见加法，梯度直接传递下来等于上一层的梯度

严格的数学表示如下图：

$$\frac{\partial J(W,b)}{\partial W^l} = \delta^l (a^{l-1})^T$$

$$\frac{\partial J(W,b,x,y)}{\partial b^l} == \delta^l$$

代码实现如下：

```
    def backward(self, top_diff):  # 反向传播的计算
        # TODO: 全连接层的反向传播，计算参数梯度和本层损失
        self.d_weight = np.dot(self.input.T, top_diff)
        self.d_bias = np.dot(np.ones([1, self.input.shape[0]]), top_diff)
        bottom_diff = np.dot(top_diff, self.weight.T)
        return bottom_diff
```

2. Softmax层反向传播

Softmax是一个激活函数，其形式如下：

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

softmax函数求导 $\frac{\partial y_i}{\partial x_j}$

(1)当 $i = j$ 时

$$\begin{aligned}
\frac{\partial y_i}{\partial x_j} &= \frac{\partial y_i}{\partial x_i} \\
&= \frac{\partial}{\partial x_i}\left(\frac{e^{x_i}}{\sum_k e^{x_k}}\right) \\
&= \frac{\left(e^{x_i}\right)'\left(\sum_k e^{x_k}\right) - e^{x_i}\left(\sum_k e^{x_k}\right)'}{\left(\sum_k e^{x_k}\right)^2} \\
&= \frac{e^{x_i} \cdot \left(\sum_k e^{x_k}\right) - e^{x_i} \cdot e^{x_i}}{\left(\sum_k e^{x_k}\right)^2} \\
&= \frac{e^{x_i} \cdot \left(\sum_k e^{x_k}\right)}{\left(\sum_k e^{x_k}\right)^2} - \frac{e^{x_i} \cdot e^{x_i}}{\left(\sum_k e^{x_k}\right)^2} \\
&= \frac{e^{x_i}}{\sum_k e^{x_k}} - \frac{e^{x_i}}{\sum_k e^{x_k}} \cdot \frac{e^{x_i}}{\sum_k e^{x_k}} \\
&= y_i - y_i \cdot y_i \\
&= y_i(1 - y_i)
\end{aligned}$$

(2)当 $i \neq j$ 时

$$
\begin{aligned}
\frac{\partial y_i}{\partial x_j} &= \frac{\partial}{\partial x_j}\left(\frac{e^{x_i}}{\sum_k e^{x_k}}\right) \\
&= \frac{\left(e^{x_i}\right)'\left(\sum_k e^{x_k}\right) - e^{x_i}\left(\sum_k e^{x_k}\right)'}{\left(\sum_k e^{x_k}\right)^2} \\
&= \frac{0 \cdot \left(\sum_k e^{x_k}\right) - e^{x_i} \cdot e^{x_j}}{\left(\sum_k e^{x_k}\right)^2} \\
&= \frac{-e^{x_i} \cdot e^{x_j}}{\left(\sum_k e^{x_k}\right)^2} \\
&= -\frac{e^{x_i}}{\sum_k e^{x_k}} \cdot \frac{e^{x_j}}{\sum_k e^{x_k}} \\
&= -y_i \cdot y_j
\end{aligned}
$$

综上所述：$\frac{\partial y_i}{\partial x_j} = \begin{cases} = y_i - y_i y_i, \text{当} i = j \\ = 0 - y_i \cdot y_j, \quad \text{当} i \neq j \end{cases}$

所以在代码实现中，我们用独热码来标志 i=j 与 i != j 的情况：

```
def backward(self):  # 反向传播的计算
        # TODO: softmax 损失层的反向传播，计算本层损失
        bottom_diff = (self.prob - self.label_onehot) / self.batch_size
        return bottom_diff
```

3. Sigmoid层反向传播

sigmod函数也是一个激活函数，其启发于人类神经元的激活形式：

其定义如下：$S(x) = \frac{1}{1+e^{-x}}$

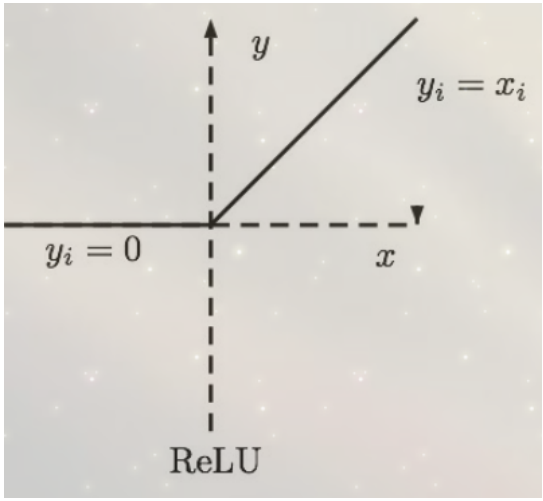
sigmoid 函数的求导就是非常难求，但是它的结果非常简单：（x-1）* x

代码实现：

```
    def backward(self, top_diff):  # 反向传播的计算
        bottom_diff = top_diff*(1-top_diff)
        return bottom_diff
```

4. ReLU层反向传播

Relu也是一个激活函数，但是它的定义比较特殊，如图：



只直观来解释就是，小于0的等于0，大于0的等于它本身。

所以它的梯度也比较特殊，本来0点无梯度，但是为了连续性，让其梯度等于0，大于0部分梯度等于1；

其代码实现:

```
    def backward(self, top_diff):  # 反向传播的计算
        # TODO：ReLU层的反向传播，计算本层损失
        bottom_diff = top_diff * (self.input >= 0.)
        return bottom_diff
```

5. 池化层反向传播

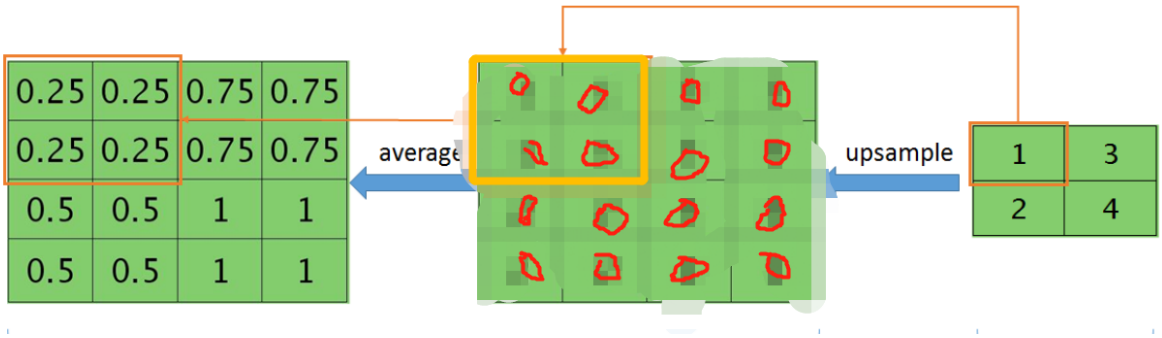
池化层，是对卷积层的一个下采样，所以在梯度回传的时候，维度是下采样之后的，肯定不对称。

那么我们就要根据池化类型，进行一个上采样。

其中:

最大池化

在前向传播的时候，记录下取最大值的掩码，在反向传播的时候，根据掩码和上一层传回的梯度，在上采样以后的梯度矩阵上，进行对应位置的填充。（如下图）



平均池化

这个实现起来不麻烦，我们实现的是平均池化。

简单来说，就是把上一层传回的梯度平均分给对应kernel区域上的所有值（如图示）：

6. 卷积层反向传播

这是最复杂的一个推导，网上所有的教程都是单通道，单卷积核的。

在多通道的时候就不是那么容易了，但是还是根据基本的三条原则：

- w是卷积核，我们可以用特征图a*top_diff来获得
- b是偏移，也是直接将上一层累加

dw求法：

这里的在求dw的时候，是多通道，多个核的；

但是其实它相当于用top_diff当做卷积核，来对特征图进行卷积操作。

w的四个维度是w（输入通道，长，宽，输出通道）

其实就等价于a的（图片数量，长，宽，输出通道）

然后此时top_diff的通道数是输出通道数，a的通道数是输入通道数。

输入通道对于w来说就是图片数量，而top_diff和a能同意起来的就是，他们图片数都是2.

所以最后我们只要在对应的维度上做了对应，然后用forward里卷积的方式来进行替换就ok。

　具体推导的式子太复杂我就不复制粘贴了。以上就是我的实现方式理解。

代码实现：

```
    def backward(self,top_diff):
        # TODO: 边界扩充
        height_out = self.input.shape[2]
        width_out = self.input.shape[3]
        height = top_diff.shape[2] + self.padding * 2
        width = top_diff.shape[3] + self.padding * 2
        top_diff_pad = np.zeros([self.input.shape[0], self.channel_out,
int(height), int(width)])
        top_diff_pad[:, :, self.padding:top_diff.shape[2]+self.padding,
self.padding:top_diff.shape[2]+self.padding] = top_diff
        self.bottom_diff = np.zeros([self.input.shape[0], self.channel_in,
int(height_out), int(width_out)])

        # TODO:先求 bottom_diff
        for idxn in range(self.input.shape[0]):
            for idxc in range(self.channel_in):
                for idxh in range(height_out):
                    for idxw in range(width_out):
                        weight=np.rot90(np.rot90(self.weight))
                        self.bottom_diff[idxn, idxc, idxh, idxw] =
np.sum(np.transpose(weight[idxc, :, :, :],(2,0,1))* top_diff_pad[idxn, :,
idxh*self.stride:idxh*self.stride+self.kernel_size,
idxw*self.stride:idxw*self.stride+self.kernel_size])
```

```
        w_height_out=self.weight.shape[1]
        w_width_out=self.weight.shape[2]
        self.d_weight=np.zeros_like(self.weight)
        top_diff_kernel_size=top_diff.shape[2]
        self.input_pad=np.transpose(self.input_pad,(1,0,2,3))
        # TODO: 再求dw
        for idxn in range(weight.shape[0]):
            for idxc in range(self.channel_out):
                for idxh in range(w_height_out):
                    for idxw in range(w_width_out):
                        self.d_weight[idxn, idxh, idxw,idxc] =
np.sum(top_diff[:, idxc, :, :] * self.input_pad[idxn, :,
idxh*self.stride:idxh*self.stride+top_diff_kernel_size,
idxw*self.stride:idxw*self.stride+top_diff_kernel_size])
        # TODO: 求出b
        self.d_bias = np.sum(top_diff,axis=(0,2,3))
        return self.bottom_diff
```

二、数据集预处理

```
### 从tensorflow下载数据集:
```

因为自动下载不成功，所以直接去下了个现成的然后放到对应的文件夹中：

```
import tensorflow.examples.tutorials.mnist.input_data as input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

然后根据要求，只进行0，1，2三个数字的二分类，所以就把0,1,2和对应的标签挑出来存放在一个文件（.npz）中，以便下次直接使用。

最后调用清洗后数据的方法是：

```
    data=np.load("mnist012.npz")
    test_x=data["test_x012"]
    test_y=data["test_y012"]
    train_x=data["train_x012"]
    train_y=data["train_y012"]
```

三、卷积神经网络搭建思路

卷积神经网络（CNN）的搭建方式一般是，前面是卷积层+池化层，用于局部特征的自动提取。后面连接上FC层作为分类器来进行分类。

各层用途概述

卷积层：受到生物视觉局部感受野启发，用于自动进行特征的提取，解决了fc层不关心像素局部近邻结构的缺点

池化层：可以有效的缩小参数矩阵的尺寸，从而减少最后连接层的中的参数数量。并且这样一个下采样的操作，让其更好的适应了不同尺寸旋转等复杂情况，增强模型鲁棒性。最重要的是，大大减少了FC层中太多参数导致参数冗余，内存浪费等问题。

扁平层：就是把卷积操作之后的多通道图片拉成1维的形式，让其能直接输入FC层能够计算。

FC层：分类器。

softmax层：在最后作为输出层，给出各个类别的分类概率。

我们的CNN模型搭建

首先要提到的是：题目要求是使用sigmoid作为激活函数，我们仅仅在fc层中间使用了sigmoid，因为对于卷积层来说，使用ReLU是一个更好的选择。具体会在高级要求部分中进行分析说明。

不算激活函数层和扁平层，我们的CNN模型一共有七层，前四层分别是一次卷积一次池化，最后三层都是fc层。

直接上代码：

```
    def build_model(self):
        # TODO: 定义VGG19 的网络结构
        print('Building vgg-19 model...')

        self.layers = {}
        self.layers['conv1_1'] = ConvolutionalLayer(3, 1, 4, 1, 1)
        self.layers['relu1_1'] = ReLULayer()
        self.layers["pool1"]=MaxPoolingLayer(2,2)
        self.layers['conv1_2'] = ConvolutionalLayer(3, 4, 8, 1, 1)
        self.layers['relu1_2'] = ReLULayer()
        self.layers['pool2'] = MaxPoolingLayer(2, 2)

        self.layers['flatten'] = FlattenLayer(input_shape=[8, 7, 7],
output_shape=[392])

        self.layers['fc4'] = FullyConnectedLayer (392, 128)
        self.layers['sig4'] = SigmoidLayer()
        self.layers['fc5'] = FullyConnectedLayer(128, 128)
        self.layers['sig5'] = SigmoidLayer()
        self.layers["fc6"]=FullyConnectedLayer(128,3)
        self.layers['softmax'] = SoftmaxLossLayer()

        self.update_layer_list = []
        for layer_name in self.layers.keys():
            if 'conv' in layer_name or 'fc' in layer_name:
                self.update_layer_list.append(layer_name)
```

四、 测试集预测

使用**sigmoid**作为FC层的激活函数，得到如下的结果：

训练集91,9375%，测试集：94.7251%

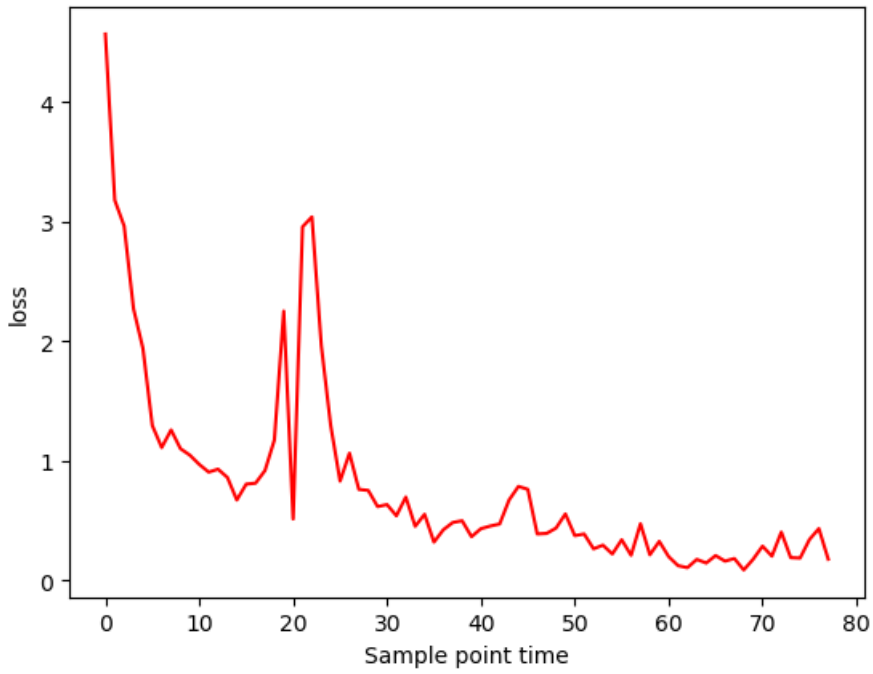
使用**Relu**作为FC层的激活函数，得到如下结果：

训练集94,4012%，测试集：95.5513%

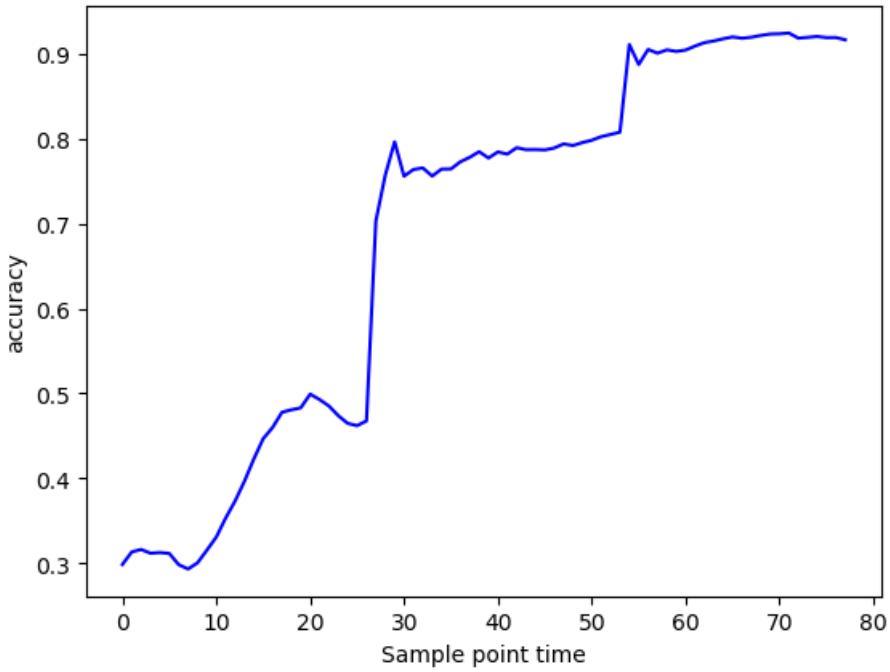
五、可视化训练情况

下面的模型都是使用ReLU作为激活函数的训练情况：

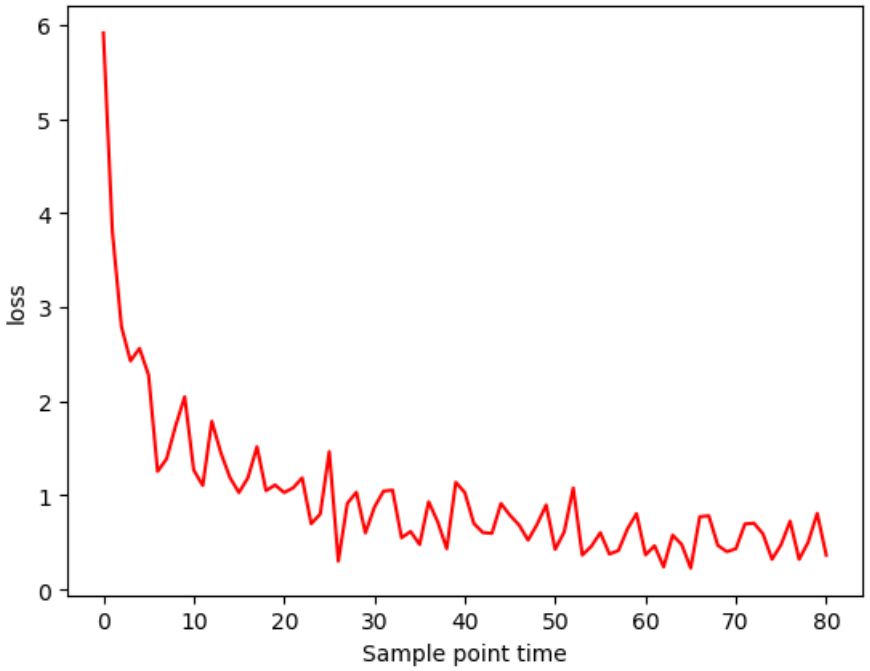
1. 训练损失



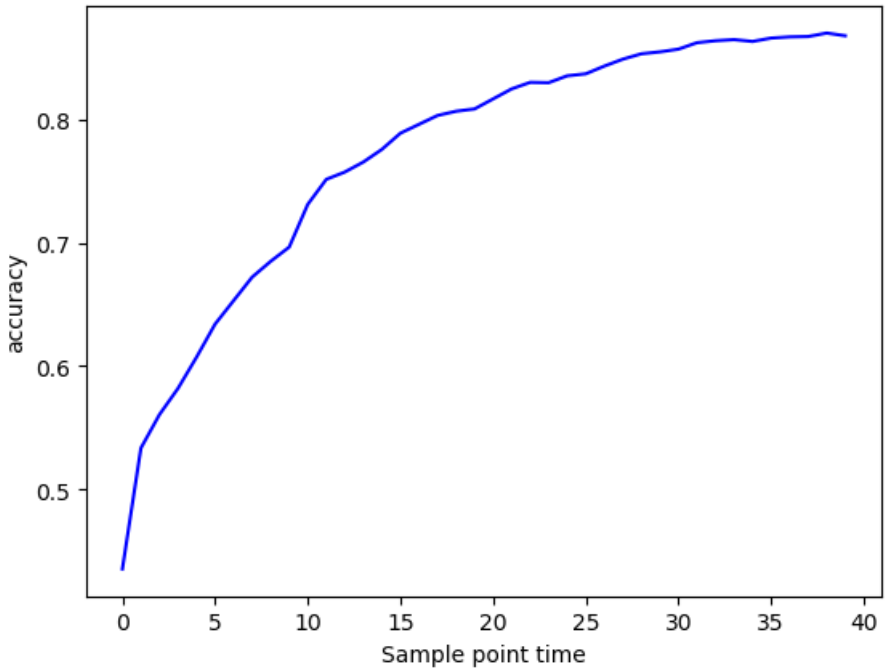
2. 训练精度


accuracy
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
10
20
30
40
50
60
70
80
Sample point time

3. 测试损失


loss
6
5
4
3
2
1
0
0
10
20
30
40
50
60
70
80
Sample point time

4. 测试精度



中级要求

> 对实验过程中可能出现的过拟合和欠拟合现象进行分析。

欠拟合和过拟合

- 欠拟合：训练了很长时间，但是在训练集上，loss值仍然很大甚至与初始值没有太大区别，而且精确度也很低，几乎接近于0，在测试集上亦如此。神经网络的欠拟合大致分为两种情况，一种是神经网络的拟合能力不足，一种是网络配置的问题。
- 过拟合：过拟合是指在模型参数拟合过程中的问题，由于训练数据包含抽样误差，训练时，复杂的模型将抽样误差也考虑在内，将抽样误差也进行了很好的拟合。具体表现就是最终模型在**训练集上效果好；在测试集上效果差**。模型泛化能力弱。

刚开始训练出现的欠拟合

刚开始训练的时候，无论训练多久loss都在1.10左右，导致我一直去寻找是不是反向传播出现了问题。

解决所有能找到的bug之后，依然无法拟合。

后来我发现是因为在随机生成w和b的时候，我设定的数据分布方差太小。每次的梯度大概在e-5次方左右，再这样的情况下，再乘以0.0005的学习率，每次学习的梯度就在e-9数量的样子出现了超参数导致的梯度消失。

终于找到问题所在，调高随即w和b的生成方差，模型就可以正常训练了。

最终模型无欠拟合或过拟合

最终的结果没有再出现欠拟合，也没有过拟合。

因为我们的网络是卷积神经网络，CNN对图像分类相比于全连接神经网咯比较不容易出现欠拟合或者过拟合的问题。

解释如下：

>

> - 图像的空间联系是局部的，就像人是通过一个局部的感受野去感受外界图像一样，每一个神经元都不需要对全局图像做感受，但是在全连接层中每个神经元都对全局图像进行了感受和学习。因为两个距离较远的像素点可能联系并不大，但是全连接层依然学习它们之间的关系，这样的话，会学习到很多冗余甚至噪音的信息。信息冗余就难免出现过拟合或者欠拟合的问题。
> - 而卷积层做的就能能够关注到局部的结构特征，相当于每个神经元只感受局部的图像区域。然后在更高层，将这些感受不同局部的神经元综合起来就可以得到全局的信息了。这样的话模型会相对更加鲁棒不容易出现欠拟合或者过拟合。
> - 池化层也对欠拟合和过拟合的问题做了优化：池化层可以有效的缩小参数矩阵的尺寸，从而减少最后连接层的中的参数数量。并且这样一个下采样的操作，让其更好的适应了不同尺寸旋转等复杂情况，增强模型泛化性能。

高级要求

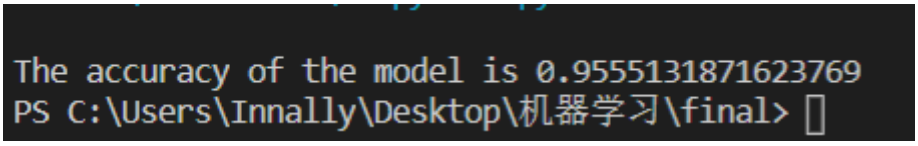
> 尝试更换其他激活函数，对比分析结果。

在这里我们对比测试了sigmoid层换成ReLU层的效果。

Sigmoid的缺点

sigmoid函数在层数加深之后，会出现梯度消失和梯度爆炸的问题，

尽管我们的层数没有很深，也还是会偶尔出现这样的求指数幂溢出的问题，说明某一层中得到了一个异常的inf值，而在我们使用Relu的时候就没有这样的情况发生。



为什么要在卷积层上使用ReLU而不是sigmoid

上面提到，大作业要求是使用sigmoid作为激活函数，但是我们没有在卷积层内使用sigmoid。

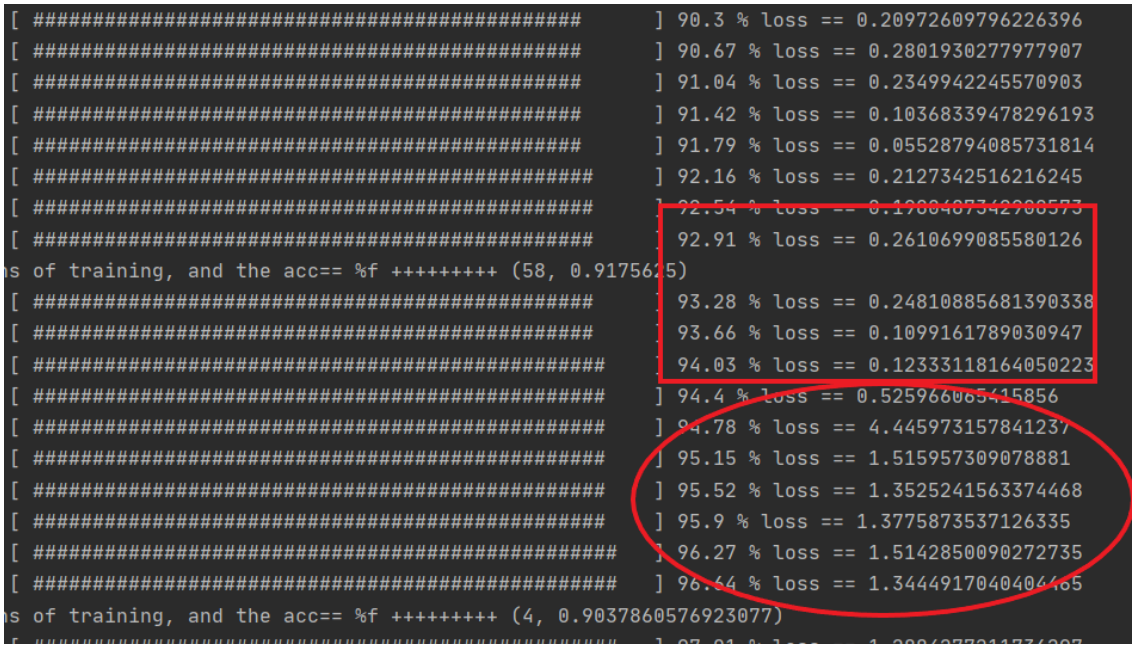
下面是我在网上找到最清楚的一个解释：

> 使用ReLU激活后，输出的矩阵产生很多0值，而通过Sigmoid激活函数后，得到的矩阵中的元素的值都是处于0到1之间，所以，ReLU的输出要比Sigmoid的稀疏程度高的多，而稀疏程度高，则意味着我们去找这些矩阵所表含的规律时就比较容易。
> 通俗点讲，当这个矩阵比较稀疏，那么它的特征会比较少，特征比较少那么它就比较好找好拟合，举个例子，比如你想点外卖，你给的特征条件是辣的，那么很好找，给个辣子鸡丁就可以了，如果你给的特征条件比较多，要辣的、甜的、酸的、冷的，这就很难找了，甚至可能找不到。
> 所以，ReLU具有稀疏化的功能，而Sigmoid的没有。

当我在卷积层上使用了sigmoid之后，收敛不了了，或许是我等得不够久，总之无法收敛，而且报了类似上面更多的warning，不是exp溢出就是multiply溢出。因此使用relu是一个明智的选择。

在FC层使用relu训练与sigmoid的对比

- 训练时间：使用sigmoid训练一个epoch的时间比relu长半分钟左右，但是差异不大
- 使用sigmoid函数出现了多次收敛后突然loss增高的问题，如下图的loss，本来已经收敛到比较低的0.2钟左右，在测试集上的准确率也到了94%，但是突然不知什么原因，loss上升到一个比较差的水平。



- 而使用Relu则没有上述问题，训练结果比较稳定，但是使用relu收敛速度较慢，sigmoid函数差不多2个epoch可以收敛到比较好的情况（虽然后面又崩坏了），Relu需要3轮才能达到差不多的准确率。
    - 训练5轮之后RELU作为激活函数的测试集预测结果，拟合效果比较不错：