# 基于 BP 算法的手写数字识别实验报告

**小组成员信息：**

何奕骁 2020212259　　杜嘉骏 2020212257　　官子鸣 2020212258

**小组成员分工如下：**

何奕骁：负责搭建基于 pytorch 的全连接神经网络模型，安排实验进度与各组员的分工。

杜嘉骏：负责基本模型、CNN 卷积神经网络的搭建和模型的优化。

官子鸣：负责资料的收集，模型的测试和部分实验报告的撰写

# 一、 实验概括

这次实验，我们基于 BP 算法，借助 numpy、matplotlib 与 pytorch 等库，分别构建了全连接神经网络与卷积神经网络来完成对手写数字的识别。其中，我们采用的是 MNIST 数据集，有 60000 个训练集与 10000 个测试集，每个样本都是一张 28＊28 像素的灰度手写数字图片。我们在自己构建全连接神经网络时遇到了训练效果不理想，难以调用 GPU 进行运算等问题，故为了提升训练效果与效率，我们借助 pytorch 机器学习库重新构建了全连接神经网络，而后，为了进一步提升训练模型的手写数字识别准确率，我们又使用 pytorch 构建了卷积神经网络。在实验过程中，我们遇到了很多问题与挫折，但是在我们的不断交流，沟通与坚持下最终解决了问题，取得了令我们比较满意的结果。

# 二、 初次构建的模型分析（模型 1）

## 1.初次构建模型简介

初次实验，我们借助 numpy 库通过矩阵运算构建了简单的基于 BP 算法的全连接神经网络，设计了 loadata 方法来读取 MNIST 数据集中的数据到内存中；然后定义了 Evaluator 评估类来对模型训练结果评测并将评测结果可视化的画图显示出来；最后，构建的 fcnn 类即为我们构建的全连接神经网络。

## 2.实验环境

实验在 win10 操作系统中运行，运行环境是 Anaconda3，编辑器为 Jupyter Notebook

必要的库是 numpy、matplotlib.pyplot 与 time。

注：因未使用 GPU 加速，纯 CPU 运行程序会因计算量过大导致可能出现卡死等状况，故使用 time 库中的 sleep 让程序在特定的位置暂停，以缓解 CPU 压力。

## 3.算法设计与方法说明

### 3.1 程序的组成

程序由以下四部分组成：

(1) 导入实验所需的库；

(2) 分别加载 MNIST 数据集中的训练样本与测试样本到内存中；

(3) 定义评估类，来对训练结果进行评测并画出图可视化评测数据；

(4) 定义全连接神经网络类并对其进行训练测试，得出训练测试的评测结果。

### 3.2 模型的算法介绍与方法说明

**(1) 模型的算法简介：**

模型采用 sigmoid 函数作为隐含层与输出层的激活函数，使用平方损失函数作为损失函数。模型可指定隐含层的层数，各层的神经元个数以及学习率与训练次数，只需要将相应的设置以初始化对象的参数的形式初始化网络对象即可。

神经元在网络中代表的只是一系列权重和阈值，所以可以使用一个向量来代表一个神经

元，所以向量组合在一起，就可以使用矩阵代表一层神经元，这样存储神经元的方式就很方便了，只需要随机初始化对应大小的矩阵即可。

在网络类中我们还添加了一个评估类对象，这个对象主要是在训练中不断记录训练结果求出训练结果中的宏平均、微平均、召回率、F 值，另外我们还加入了小类的 4 个评估标准的计算，并且在最后可视化训练结果，使得更容易分析结果以改进网络。

在训练的过程中我们选用训练的方法是小批量训练，主要思想是随机从训练数据集中选择指定批量的样本，然后对每个样本求出网络的反向梯度之和，使用梯度和的平均作为网络的梯度更新网络的权值和阈值，这样训练的优点是可以增加大批量训练的效率，归一化了损失情况，增大学习率的稳定性，平均掉了数据集的噪声。在激励函数方面，我们选择的是书本上介绍的 sigmoid 函数。在 numpy 的支持下可以很容易地实现矩阵的运算，这样经过前向传播和 BP 反向传播就可以不断训练网络神经元的权值和阈值，不断提升准确率。

最后我们使用了测试集对网络的训练结果进行了评估，在评估的过程中我们只是用了前向传播对测试数据进行预测，根据宏平均、微平均、召回率、F 值的计算方法求出对应的值，最后对在测试集上的表现进行可视化和相应的打印。

以上就是模型的训练和评估算法。

**(2) 模型的方法说明：**

```
__init__(self,layers:list)->None:
```

这是模型的构造方法，使用 layers 初始化网络的每层的神经元个数以及神经元权重，神经元激活阈值参数。layers 列表从低到高各项为从输入层到输出层的神经元个数。

```
def sigmoid(self,x:np.ndarray)->np.ndarray:
def dsigmoid(self,x:np.ndarray)->np.ndarray:
```

这是模型的激活函数与其导数，采用的是 sigmoid 函数，输入的是需要计算的矩阵，输出其每一项经函数计算后的结果矩阵。

```
def backward(self,x:np.ndarray,y:np.ndarray)->tuple:
```

这是反向传播函数，传入特定样本输入数据构成矩阵 x 与它们的标签 y，进行前向传播，损失函数计算，反向传播等过程，返回神经元权重与神经元激活阈值参数对于损失函数的偏导 dw 与 db。

```
def predict(self,x:np.ndarray)->int:
```

这是模型的预测方法，传入特定样本输入数据构成的矩阵，经过前向传播得到输出层的输出矩阵，并将其矩阵中的最大值所在行作为手写数字的识别结果并返回。

```
def update(self,mdata:list,rate:float)->None:
```

这是更新参数的方法，传入每步训练的样本集 mdata 与学习率 rate(实际学习率为 rate/len(mdata))，调用 backward 方法求出各参数对损失函数的偏导，并以此进行参数更新，不断训练模型。

```
def startrain(self,data,label,batch,loop,rate,test1 =[],test2 =[])->None:
```

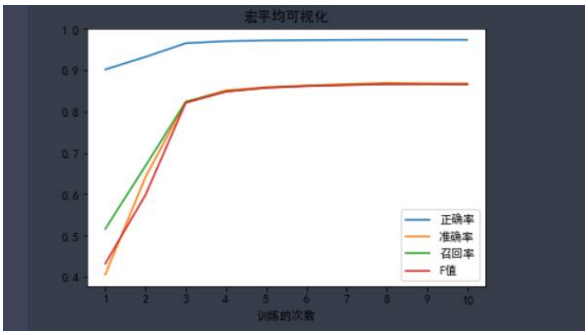
调用该方法开始训练，传入的 data，label 分别为训练集与训练集的标签，batch，loop 与 rate 分别为训练的步长，训练次数以及学习率(实际学习率为 rate/ batch)，test1 与 test2 为测试集以及其标签。

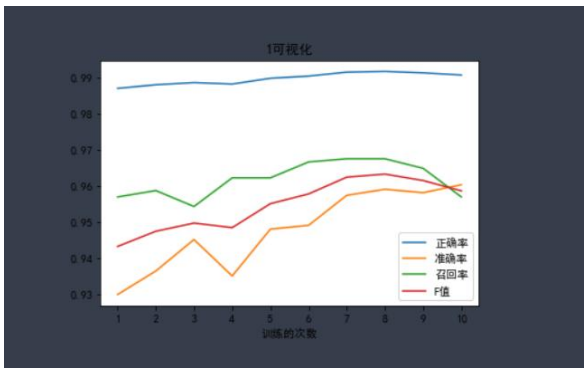
```
def startest(self,test:list,label:list)->float:
```

调用该方法进行测试，传入测试集以及其标签，对模型的训练结果进行评估，并可视化的显示出评估结果，如下图(部分)所示：

```
带有测试集的训练开始. . . . . .

第 1 次训练开始. . . . . .
--------第 1 次训练结束.
----------------------------总体准确率是:   0.699

第 2 次训练开始. . . . . .
--------第 2 次训练结束.
----------------------------总体准确率是:   0.8232
```

```
第 10 次训练开始. . . . . .
--------第 10 次训练结束.
----------------------------总体准确率是:   0.8943

结果可视化. . . . . .
```



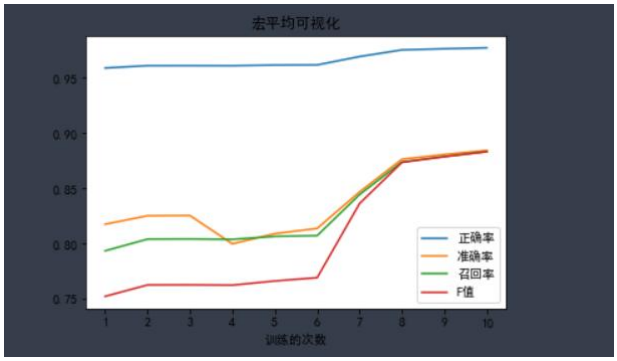



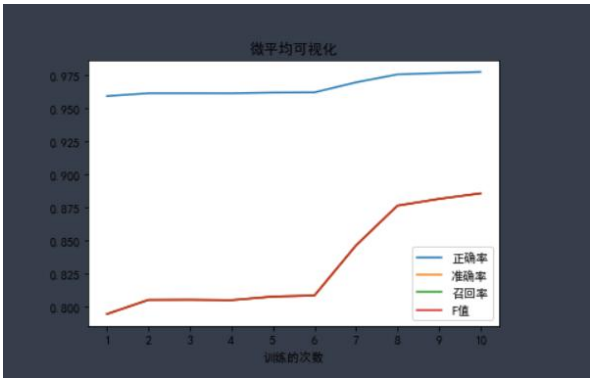

## 4. 模型测试与评估

一层 40 个神经元的隐含层，步长为 64，学习率为 8（真正改变参数的 rate 为学习率/步长，下同），训练十次，结果如下（篇幅受限，仅展示整体的评估图像）：

（模型的总体准确率为测试集中所有被正确识别的样本数/测试集的总样本数）

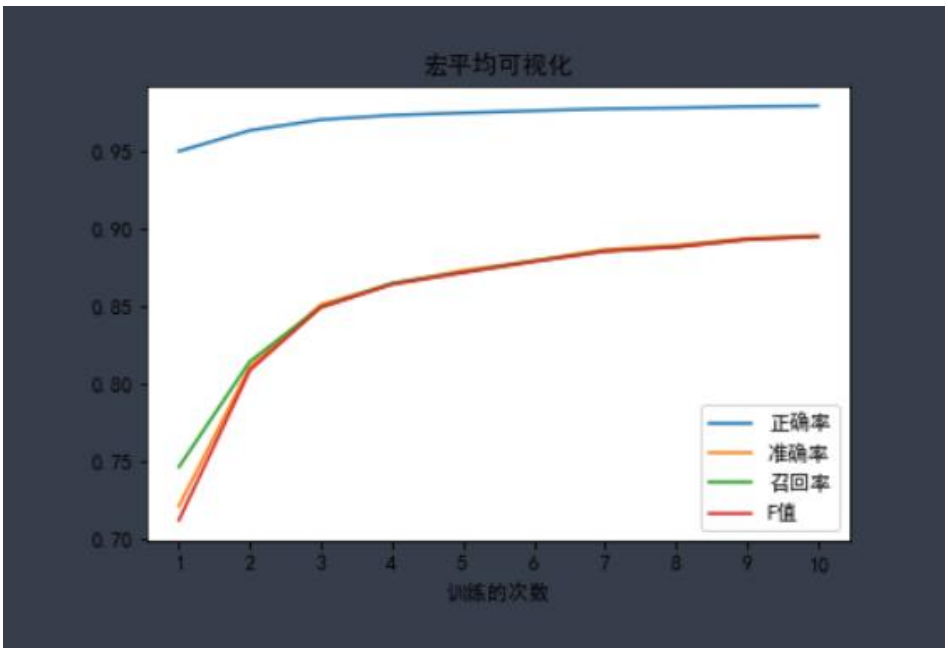
```
第 10 次训练开始. . . . . .
--------第 10 次训练结束.
----------------------------总体准确率是:   0.8853
```

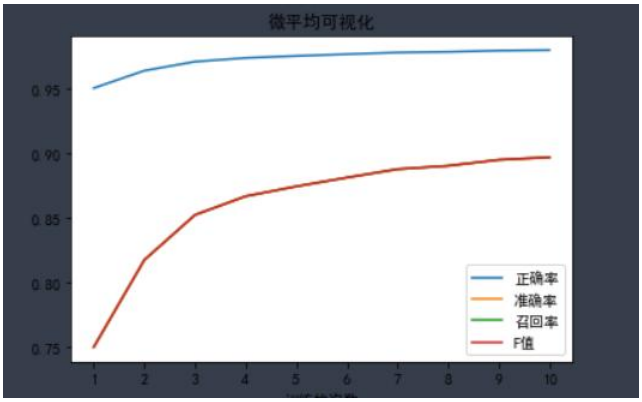



一层 40 个神经元的隐含层，步长为 64，学习率为 1，训练十次，结果如下：

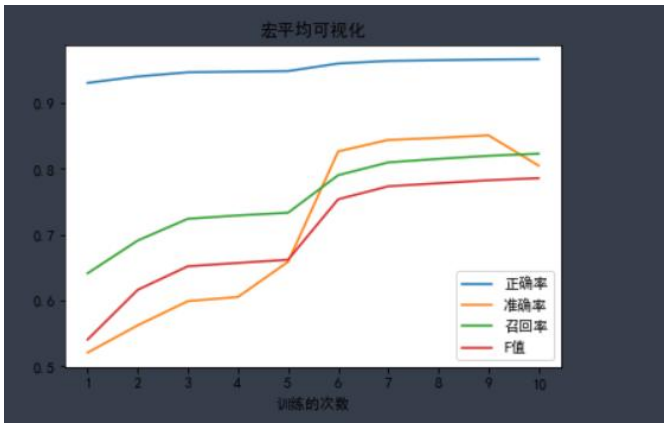
```
第 10 次训练开始. . . . . .
--------第 10 次训练结束.
------------------------------总体准确率是:    0.8967
```






一层 80 个神经元的隐含层，步长为 64，学习率为 1，训练十次，结果如下：

```
第 10 次训练开始. . . . . .
--------第 10 次训练结束.
----------------------------总体准确率是:   0.8333
```




**对评估结果的分析：**

除了上面展示的部分训练结果外，我们还改变步长，隐含层层数与神经元个数等参数，测试了多次，然而，结果不尽人意，总体的准确率很难突破百分之九十，但是对于特殊的数字比如“1”、“7”等的识别率是非常高的，但是对于“5”、“6”、“8”、“9”等的识别率总是不断的上下浮动，时而 95%多，时而到 70%多，而无法达到所有的数字识别率都同时很高的情况。经过我们多次训练的测试，最好的总体准确率达到 89.67%，在达到峰值后，准确率就会由于过拟合问题不断下降，而且，同一参数，训练多次的结果也有较大的差异；另外，我们电脑的 CPU 在训练多层隐含层时难以有较好的表现。为了提升我们模型的准确度以及训练效率，我们决定改用 pytorch 机器学习库来构建全连接神经网络，并使用 CUDA 进行 GPU 加速，提升训练效率。

在此处我们尝试了很多的小方法来提升我们的正确率，比如对于正确率很早期很难提高的数字，我们采用预训练的方法来对随机化的参数进行一个偏移，从而使得网络在开始训练后能够更快的提升这部分数字的识别准确率，但是经过实验，我们知道了这种方法是错误的，因为加上了预训练就会使得网络的权值在很大程度上不在随机化，从而使得网络对于预训练的数字有很强的敏感性，而降低了其他数字的准确率。

## 三、 pytorch 构建的全连接神经网络模型分析（模型 2）

### 1. 模型 2 改进和算法优化方法

（1）在运行效率的改进上我们调用 CUDA 提供的接口，将构建模型与数据迭代器等放在 GPU 上，利用 GPU 强大的运算能力提升训练的效率。

（2）在数据集的加载方面，我们使用了 torchvision 中提供的 MNIST API，可以在加载数据的同时将数据进行归一化和标准化；按照小样本的方式将数据集切分成小样本集；并且转换成 torch 里的 tensor 张量数据类型，从而利用 torch 内置的优化器对运算进行优化。

（3）在网络的构建中的我们采用了改进激活函数和损失函数的方式进行优化。其中激活函数改为 relu 激活函数，这样会避免 sigmoid 激活函数的梯度消失问题，损失函数我们使用了交叉熵损失函数，这样的损失函数计算梯度时更方便，并且由于最小化交叉熵损失函数对应的是极大似然估计，所以优化的结果更精确。

（4）在反向传播中我选择的是 Adam 优化器进行自动求导和更新参数。Adam 优化器能够对不同的参数调整不同的学习率，具体表现是对频繁变化的参数以更小的步长进行更新，对变化不明显的参数以更大的步长进行更新，能够更好的利用梯度信息，比标准的 SGD 算法更有效的收敛。

### 2. 模型 2 实验环境

实验在 win10 操作系统中运行，运行环境是 Anaconda3，编辑器为 Jupyter Notebook，必要的库是 numpy、matplotlib.pyplot、torchvision、PIL 与 time 等，另外还需要安装 CUDA。

### 3. 模型 2 测试与评估

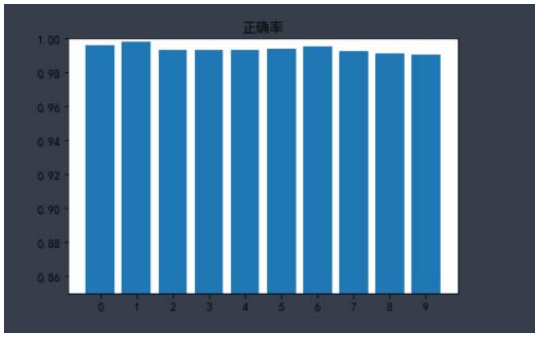
两层每层有 40 个神经元的隐含层，每个批量样本数为 64，学习率为 0.001，训练十次，结果如下：

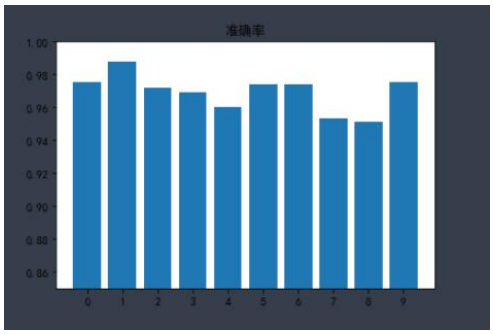
训练集的训练结果：

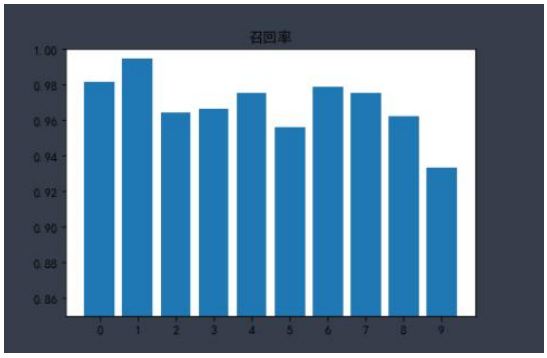
```
第10次训练开始. . . .
-----------------------第10次训练结束, 总体准确率是 0.9946
59674 60000
```

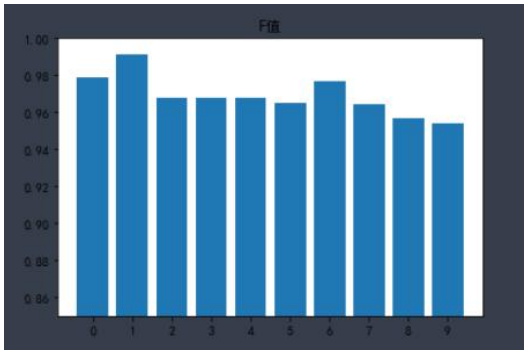
测试集评估结果：

```
损失函数总和: 0.1706, 正确率: 9693/10000 (96.93%)

宏平均:

--------正确率:0.9938600000000001

--------准确率:0.9691855899532239

--------召回率:0.9689082387171863

-------- F值  :0.9689671187313392

微平均:

--------正确率:0.9938599999999999

--------准确率:0.9692999999999999

--------召回率:0.9692999999999999

-------- F值  :0.9692999999999999
```



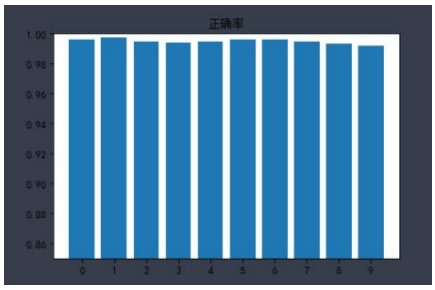



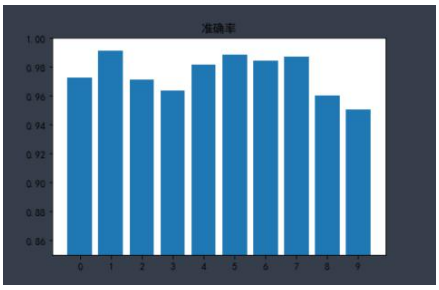

而将每层隐含层的神经元个数改为 80，则：
训练集的训练结果：

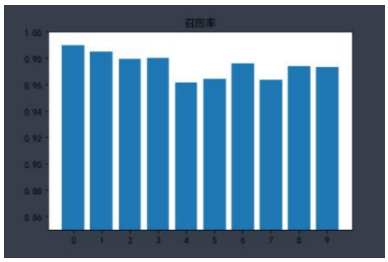
```
第10次训练开始. . . .
------------------------第10次训练结束, 总体准确率是 0.991
59462 60000
```

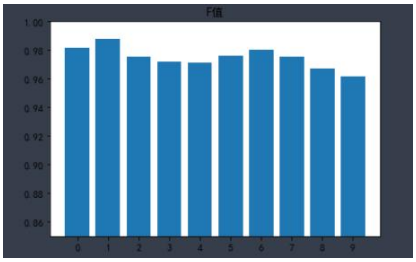
测试集评估结果：

```
损失函数总和: 0.0914, 正确率: 9750/10000 (97.50%)

宏平均:

--------正确率:0.9950000000000001

--------准确率:0.9751420398948996

--------召回率:0.9747669063000199

-------- F值  :0.9748786160455962

微平均:

--------正确率:0.995

--------准确率:0.975

--------召回率:0.975

-------- F值  :0.975
```




由上评估结果可知，改用 pytorch 框架后我们的模型对手写数字识别的准确率大幅提升，并且使用 GPU 加速后训练的效率也有了大幅度的提升。目前，我们的模型能对测试集的识别准确度达到 0.97 左右，这已经是一个令人比较满意的结果了。但是，查阅相关资料后，我们得知，卷积神经网络在图像识别领域有着广泛的应用，并且其具有更好的稳定性，因此我们开始构建 CNN 神经网络，以进一步提升我们模型的识别准确率。

## 四、pytorch 构建的卷积神经网络模型分析（模型 3）

### 1. 模型 3 改进和算法优化方法

卷积神经网络（CNN）是一种前馈神经网络，它由若干卷积层和池化层组成，在图像识别方面中 CNN 的识别效果很好。

CNN 基本结构是若干卷积层、池化层交替连接，最后加上全连接层及输出层构成，在实验中我们使用的是卷积层+池化层+卷积层+三层全连接网络的结构，其中卷积层主要是用来进行特征提取；池化层对特征提取后的特征图进行压缩，将特征图的维度降低，等同于降低了模型的复杂度，另一方面，压缩也是在提取主要特征，放弃次要的特征；最后的全连接层主要是对提取的特征进行分类，得到分类的结果。由于卷积层的维度和计算量相较于全连接层都很大，所以我们主要在卷积层进行计算的优化，在全连接层进行参数的优化。

对于全连接层的优化，我们使用的是交叉熵损失函数，使用了通用的小样本学习的方法，并且使用引入 Momentum 的 SGD 算法进行梯度下降，它引入了动量原则，基本的思想是和惯性相似，当前权值的改变会收到上一次权值改变的影响，收敛过程中权值的摆动更大。

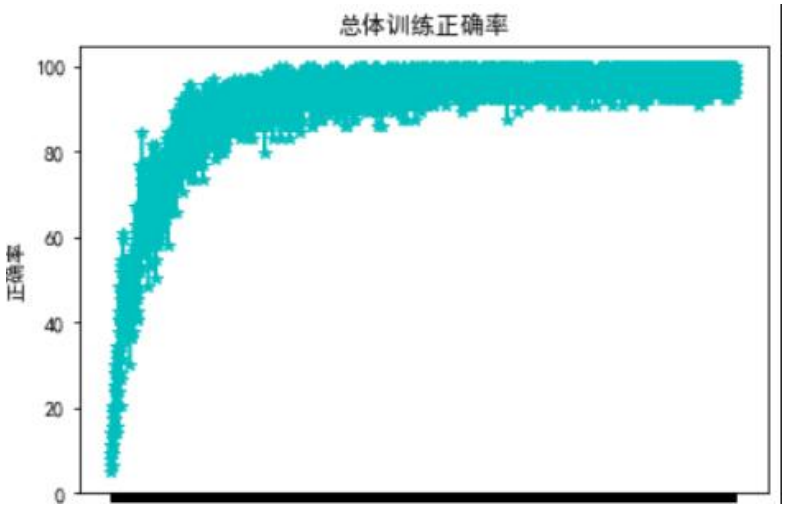
而对于卷积层的计算方面的优化，我们使用了 GPU 提供的高速矩阵运算功能，这样使得训练过程更加高效。

### 2. 模型 3 的实验环境

实验在 win10 操作系统中运行，运行环境是 Anaconda3，编辑器为 Jupyter Notebook，必要的库是 numpy、matplotlib.pyplot、torchvision、PIL 等，另外还需要安装 CUDA。

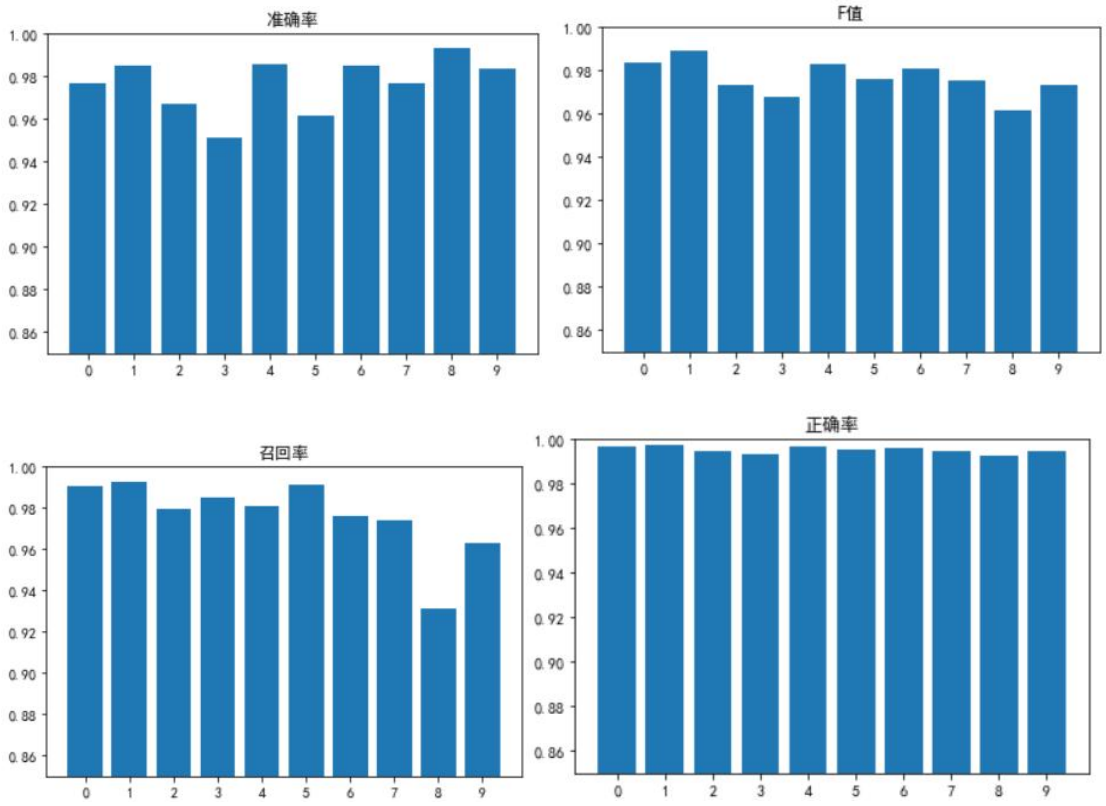
## 3. 模型 3 的测试与评估

我们按照“卷积层+池化层+卷积层+三层全连接网络”的结构来初始化了卷积神经网络，并且我们设置梯度下降的步长是 0.001，训练 5 次，得到的结果如下：




```
第5次训练开始. . . . . .

--------------------------------第5次训练结束，总体的正确率是98.4375
```

```
总体的正确率是: 0.9766

宏平均:

--------正确率:0.9953199999999999

--------准确率:0.9767360438031935

--------召回率:0.9764519253853775

-------- F值  :0.9764179299797394

微平均:

--------正确率:0.9953200000000001

--------准确率:0.9766

--------召回率:0.9766

-------- F值  :0.9766

测试结束 . . . . . .
```




由上结果可知，使用卷积神经网络不仅使得网络的准确率得到大幅度的提升，而且网络的稳定性很好，在训练的后期过程中，网络的准确率丝毫没有降低，能够很好的避免过拟合问题，具有良好的稳定性。

## 四、 对实验的总结、讨论和思考

在实验中我们首先通过实践所学，自己搭建基于 BP 算法的神经网络，但是面对其复杂的优化方法，我们转向 pytorch 库使用其中的优化方法进行改进，从而大幅度提升训练的准确率，最后我们又使用了卷积神经网络进一步对手写数字识别的模型进行了优化。在实验中我们对 BP 算法有了更深入的理解，并且收获了自己搭建神经网络的经验和教训，了解到了深度学习的强大与魅力。

起初，因为缺少经验，我们搭建的神经网络模型进行手写数字识别的效果不尽如人意，但是在查阅了相关资料，组员间进行交流沟通后我们采用 pytorch 改善了全连接神经网络，提升了我们模型手写数字识别的能力，最后我们使用了 CNN，进一步提升了我们模型手写数字识别的效果，较为完美的完成了这次实验。

经过这次实验，我们的自学能力和编程能力得到了提升，在模型设计和参数选择方面也有了一定的思考。我们认为，人工智能和深度学习必将成为未来的发展趋势，我们会结合本次实验中所学的知识，未来继续在这条道路上前进。