**《数据挖掘》实验报告**

|  |  |
| --- | --- |
| 实验名称： | MNIST手写数字识别 |
| 院（系）： | 电子信息与通信学院 |
| 专业班级： |  |
| 姓名： |  |
| 学号： |  |
| 时间： | 2021.05.18 |
| 实验成绩： |  |
| 指导教师： | 江国星 |

2021年 05月18日

目录

[一、 实验目的 1](#_Toc72326497)

[二、 实验环境 1](#_Toc72326498)

[三、 实验原理 1](#_Toc72326499)

[1. Keras API： 1](#_Toc72326500)

[2. PaddlePaddle API： 1](#_Toc72326501)

[3. Pytorch API： 2](#_Toc72326502)

[4. MNIST数据集： 2](#_Toc72326503)

[5. Softmax： 3](#_Toc72326504)

[6. LeNet: 4](#_Toc72326505)

[四、 具体实现 6](#_Toc72326506)

[1. Keras基于MLP实现： 6](#_Toc72326507)

[2. PaddlePaddle使用LeNet实现： 7](#_Toc72326508)

[3. Pytorch使用CNN实现： 10](#_Toc72326509)

[五、 实验小结 13](#_Toc72326510)

# 实验目的

1．掌握卷积神经网络CNN的基本原理；

2．利用LeNet实现手写数字识别。

3. 以MNIST手写数字识别为课题，研究基本数据挖掘方法的应用。

# 实验环境

### 操作系统：Windows 10

### 开发IDE：PyCharm 2021.1.1 (Professional Edition)

### 深度学习API：Keras-gpu 2.3.1

### Paddle 2.0.2

**Pytorch 1.8.1**

# 实验原理

## Keras API：

Keras 是一个用 Python 编写的高级神经网络 API，它能够以 TensorFlow, CNTK, 或者 Theano 作为后端运行。Keras 的开发重点是支持快速的实验。能够以最小的时延把想法转换为实验结果，是做好研究的关键。

## PaddlePaddle API：

PaddlePaddle 是起源于百度的开源深度学习平台。它是简单易用的：你可以通过简单的十数行配置搭建经典的神经网络模型；它也是高效强大的：PaddlePaddle可以支撑复杂集群环境下超大模型的训练，令你受益于深度学习的前沿成果。在百度内部，已经有大量产品线使用了基于PaddlePaddle的深度学习技术。

我认为，虽然现在PaddlePaddle的市场占有率不高，但作为我们国产的深度学习平台，在我们国内广大开发者的支持与帮助下，肯定会发展的更好。

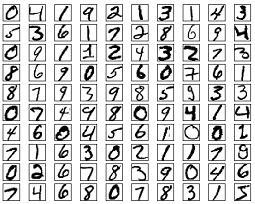
## Pytorch API：

Pytorch是torch的python版本，是由Facebook开源的神经网络框架，专门针对 GPU 加速的深度神经网络（DNN）编程。Torch 是一个经典的对多维矩阵数据进行操作的张量（tensor ）库，在机器学习和其他数学密集型应用有广泛应用。与Tensorflow的静态计算图不同，pytorch的计算图是动态的，可以根据计算需要实时改变计算图。但由于Torch语言采用 Lua，导致在国内一直很小众，并逐渐被支持 Python 的 Tensorflow 抢走用户。作为经典机器学习库 Torch 的端口，PyTorch 为 Python 语言使用者提供了舒适的写代码选择。

## MNIST数据集：

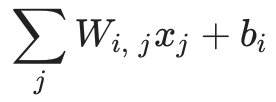
该数据集包含60,000个用于训练的示例和10,000个用于测试的示例。这些数字已经过尺寸标准化并位于图像中心，图像是固定大小(28x28像素)，其值为0到1。为简单起见，每个图像都被平展并转换为784(28 \* 28)个特征的一维numpy数组。

概览：

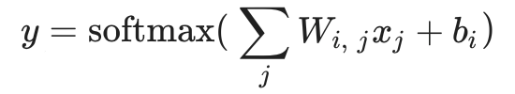


## Softmax：

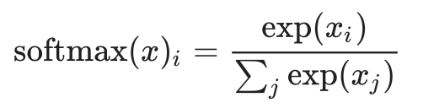
对于一个输入的数据点xi={x1,x2,...,xn}，我们可以仿照线性回归的方法给它找到一组权重Wi 和一个偏置 b 把它映射到因变量空间，线性的映射可以写为



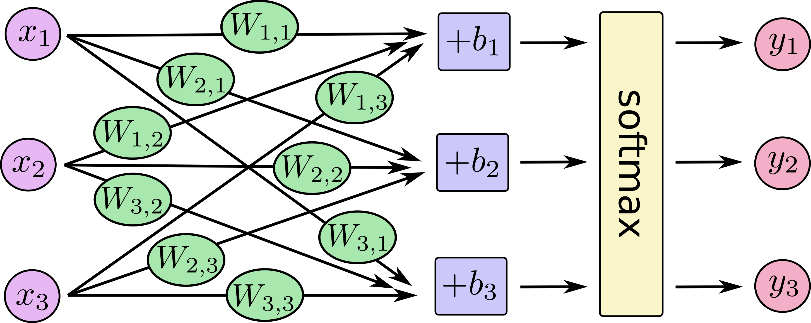
但是我们希望得到的是关于分类判别的一个概率值，所以要加一个softmax函数把上面的线性组合“扭曲”（投影）到新的取值范围内，即（其中y是一个表示各种分类结果可能性的向量）



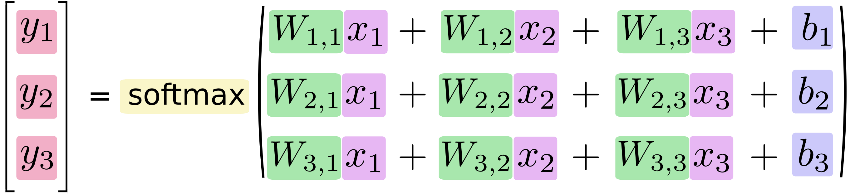
其中，softmax函数的形式如下



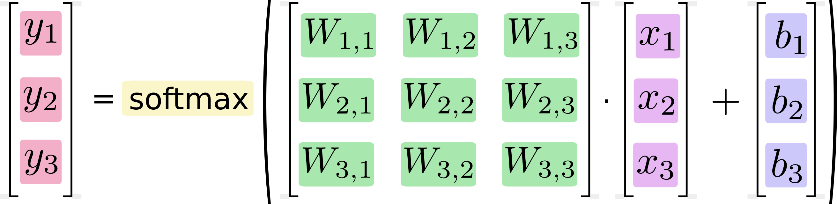
如果你把全连接的前馈神经网络图画出来，那么它的形式如下（假设每个数据点的特征向量维度是3，而且分类结果也只有3种，注意特征向量的维度和分类结果的种类可以不同。在我们本次实验中，输入图像的向量大小是724，但是分类结果只有10种）：



如果把所有的方程合在一起写出方程组的形式，则有：

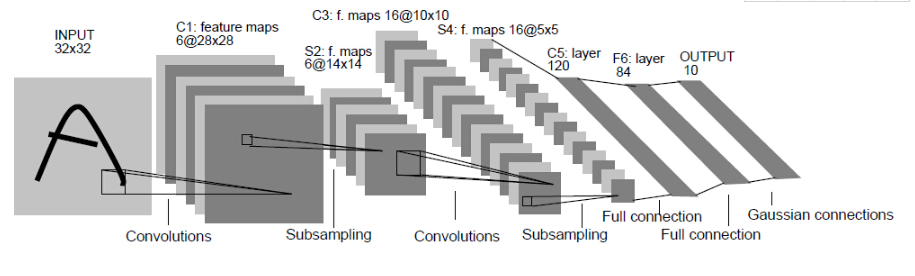


或者也可以用矩阵的形式来表示：



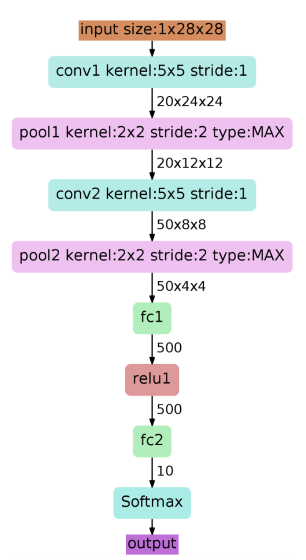
上面的公式相当于 y = softmax(Wx + b)，其实下面我们在编程实现的使用使用的是y = softmax(xW + b)。

## LeNet:



LeNet是卷积神经网络的祖师爷LeCun在1998年提出，用于解决手写数字识别的视觉任务。自那时起，CNN的最基本的架构就定下来了：卷积层、池化层、全连接层。如今各大深度学习框架中所使用的LeNet都是简化改进过的LeNet-5（-5表示具有5个层），和原始的LeNet有些许不同，比如把激活函数改为了现在很常用的ReLu。

LeNet-5跟现有的conv->pool->ReLU的套路不同，它使用的方式是conv1->pool->conv2->pool2再接全连接层，但是不变的是，卷积层后紧接池化层的模式依旧不变。



以上图为例，对经典的LeNet-5做深入分析：

首先输入图像是单通道的28\*28大小的图像，用矩阵表示就是[1,28,28]

* 第一个卷积层conv1所用的卷积核尺寸为5\*5，滑动步长为1，卷积核数目为20，那么经过该层后图像尺寸变为24，28-5+1=24，输出矩阵为[20,24,24]。
* 第一个池化层pool核尺寸为2\*2，步长2，这是没有重叠的max pooling，池化操作后，图像尺寸减半，变为12×12，输出矩阵为[20,12,12]。
* 第二个卷积层conv2的卷积核尺寸为5\*5，步长1，卷积核数目为50，卷积后图像尺寸变为8,这是因为12-5+1=8，输出矩阵为[50,8,8].
* 第二个池化层pool2核尺寸为2\*2，步长2，这是没有重叠的max pooling，池化操作后，图像尺寸减半，变为4×4，输出矩阵为[50,4,4]。
* pool2后面接全连接层fc1，神经元数目为500，再接relu激活函数。
* 再接fc2，神经元个数为10，得到10维的特征向量，用于10个数字的分类训练，送入softmaxt分类，得到分类结果的概率output。

# 具体实现

## Keras基于MLP实现：

### 加载数据：

载入并准备好 MNIST 数据集。将样本从整数转换为浮点数：

|  |
| --- |
| 1. import tensorflow as tf 2. mnist = tf.keras.datasets.mnist 3. (x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data() 4. x\_train, x\_test = x\_train / 255.0, x\_test / 255.0 |

### 模型建立：

使用MLP多层感知机，将模型的各层堆叠起来，以搭建 tf.keras.Sequential 模型。为训练选择优化器和损失函数：

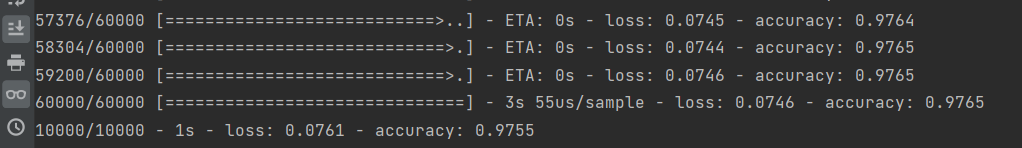
|  |
| --- |
| 1. model = tf.keras.models.Sequential([ 2. tf.keras.layers.Flatten(input\_shape=(28, 28)), 3. tf.keras.layers.Dense(128, activation='relu'), 4. tf.keras.layers.Dropout(0.2), 5. tf.keras.layers.Dense(10, activation='softmax') 6. ]) 7. model.compile(optimizer='adam', 8. loss='sparse\_categorical\_crossentropy', 9. metrics=['accuracy']) |

### 训练并验证模型：

使用fit API:

|  |
| --- |
| 1. model.fit(x\_train, y\_train, epochs=5) 2. model.evaluate(x\_test,  y\_test, verbose=2) |

结果如下所示：



可见，多层感知机已经可以获得较好的结果。

## PaddlePaddle使用LeNet实现：

### 加载数据：

用飞桨框架自带的 paddle.vision.datasets.MNIST 完成mnist数据集的加载。

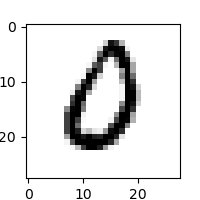
|  |
| --- |
| 1. from paddle.vision.transforms import Compose, Normalize 2. transform = Compose([Normalize(mean=[127.5], 3. std=[127.5], 4. data\_format='CHW')]) 5. *# 使用transform对数据集做归一化* 6. print('download training data and load training data') 7. train\_dataset = paddle.vision.datasets.MNIST(mode='train', transform=transform) 8. test\_dataset = paddle.vision.datasets.MNIST(mode='test', transform=transform) 9. print('load finished') |

这里尝试取训练集中的第666条数据看一下：

|  |
| --- |
| 1. import numpy as np 2. import matplotlib.pyplot as plt 3. train\_data0, train\_label\_0 = train\_dataset[666][0],train\_dataset[666][1] 4. train\_data0 = train\_data0.reshape([28,28]) 5. plt.figure(figsize=(2,2)) 6. plt.imshow(train\_data0, cmap=plt.cm.binary) 7. print('train\_data0 label is: ' + str(train\_label\_0)) |

结果如下：





可见数据加载成功。

### 模型建立：

我们直接用paddle.nn下的API，如Conv2D、MaxPool2D、Linear完成LeNet的构建。

代码如下：

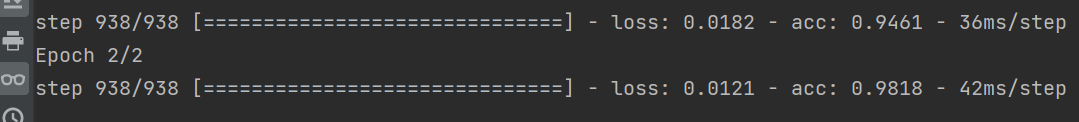
|  |
| --- |
| 1. import paddle 2. import paddle.nn.functional as F 3. class LeNet(paddle.nn.Layer): 4. def \_\_init\_\_(self): 5. super(LeNet, self).\_\_init\_\_() 6. self.conv1 = paddle.nn.Conv2D(in\_channels=1, out\_channels=6, kernel\_size=5, stride=1, padding=2) 7. self.max\_pool1 = paddle.nn.MaxPool2D(kernel\_size=2,  stride=2) 8. self.conv2 = paddle.nn.Conv2D(in\_channels=6, out\_channels=16, kernel\_size=5, stride=1) 9. self.max\_pool2 = paddle.nn.MaxPool2D(kernel\_size=2, stride=2) 10. self.linear1 = paddle.nn.Linear(in\_features=16\*5\*5, out\_features=120) 11. self.linear2 = paddle.nn.Linear(in\_features=120, out\_features=84) 12. self.linear3 = paddle.nn.Linear(in\_features=84, out\_features=10) 13. def forward(self, x): 14. x = self.conv1(x) 15. x = F.relu(x) 16. x = self.max\_pool1(x) 17. x = F.relu(x) 18. x = self.conv2(x) 19. x = self.max\_pool2(x) 20. x = paddle.flatten(x, start\_axis=1,stop\_axis=-1) 21. x = self.linear1(x) 22. x = F.relu(x) 23. x = self.linear2(x) 24. x = F.relu(x) 25. x = self.linear3(x) 26. return x |

### 模型训练：

通过paddle提供的Model 构建实例，使用封装好的训练与测试接口，快速完成模型训练与测试。

|  |
| --- |
| 1. from paddle.metric import Accuracy 2. model = paddle.Model(LeNet())   *# 用Model封装模型* 3. optim = paddle.optimizer.Adam(learning\_rate=0.001, parameters=model.parameters()) 4. *# 配置模型* 5. model.prepare( 6. optim, 7. paddle.nn.CrossEntropyLoss(), 8. Accuracy() 9. ) 10. *# 训练模型* 11. model.fit(train\_dataset, 12. epochs=2, 13. batch\_size=64, 14. verbose=1 15. ) |

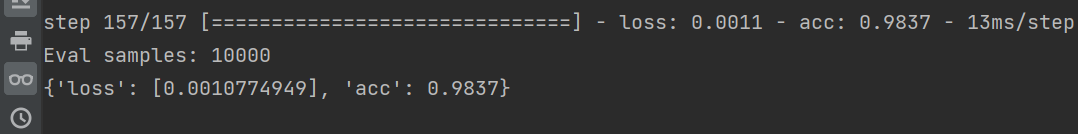
训练情况如下所示：



### 模型测试：

使用 Model.evaluate 来预测模型：

结果如下所示：



## Pytorch使用CNN实现：

参考Pytorch官方的实例。

### 加载数据：

加载训练集和测试集，pytorch 自带有 MNIST 数据集。 并且转为我们需要的格式。

|  |
| --- |
| 1. dataset1 = datasets.MNIST('../data', train=True, download=True, 2. transform=transform) 3. dataset2 = datasets.MNIST('../data', train=False, 4. transform=transform) 5. train\_loader = torch.utils.data.DataLoader(dataset1,\*\*train\_kwargs) 6. test\_loader = torch.utils.data.DataLoader(dataset2, \*\*test\_kwargs) |

### 模型建立：

建立CNN网络：

|  |
| --- |
| 1. def forward(self, x): 2. x = self.conv1(x) 3. x = F.relu(x) 4. x = self.conv2(x) 5. x = F.relu(x) 6. x = F.max\_pool2d(x, 2) 7. *#最大池化，相比平均池化特征更明显，24\*24\*64->12\*12\*64* 8. x = self.dropout1(x) 9. x = torch.flatten(x, 1) 10. x = self.fc1(x) 11. x = F.relu(x) 12. x = self.dropout2(x) 13. x = self.fc2(x) 14. output = F.log\_softmax(x, dim=1) 15. *#log\_softmax激活函数一定要搭配 nll\_loss() 损失函数处理分类问题* 16. *#如果直接输出x，则可以用 CrossEntropyLoss() 损失函数处理分类问题，见莫凡教程* 17. *#两者区别不大 CrossEntropyLoss() 在内部求了 log\_softmax(x,dim =1)。* 18. return output |

### 模型训练：

|  |
| --- |
| 1. def train(args, model, device, train\_loader, optimizer, epoch): 2. model.train() 3. *#注意和test时的 model.eval()区分，这里启用 BatchNormalization 和 Dropout* 4. for batch\_idx, (data, target) in enumerate(train\_loader): 5. *# enumerate()将数据组合为可索引的序列，并返回下标* 6. data, target = data.to(device), target.to(device) 7. *# 将输入数据，期望值传入到cuda中，如果用gpu必须写，不然和net无法连通* 8. optimizer.zero\_grad() 9. *#梯度归零* 10. output = model(data) 11. *#计算输出值，整个batch的输出值，返回一个tensor* 12. loss = F.nll\_loss(output, target) 13. *#计算loss，具体哪种loss对应哪种问题自行查阅* 14. loss.backward() 15. *#误差的反向计算，更新梯度* 16. optimizer.step() 17. *#由算出来的梯度进行优化* 18. if batch\_idx % args.log\_interval == 0: 19. *#训练时的输出* 20. print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format( 21. epoch, batch\_idx \* len(data), len(train\_loader.dataset), 22. 100. \* batch\_idx / len(train\_loader), loss.item())) 23. *# .item()取torch变量的数据值* 24. if args.dry\_run: 25. break |

### 模型测试：

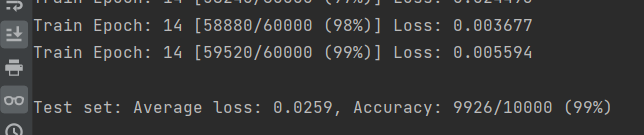
|  |
| --- |
| 1. def test(model, device, test\_loader): 2. model.eval() 3. test\_loss = 0 4. correct = 0 5. with torch.no\_grad(): 6. for data, target in test\_loader: 7. data, target = data.to(device), target.to(device) 8. output = model(data) 9. test\_loss += F.nll\_loss(output, target, reduction='sum').item()  *# sum up batch loss* 10. pred = output.argmax(dim=1, keepdim=True)  *# get the index of the max log-probability* 11. correct += pred.eq(target.view\_as(pred)).sum().item() 12. test\_loss /= len(test\_loader.dataset) 13. print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format( 14. test\_loss, correct, len(test\_loader.dataset), 15. 100. \* correct / len(test\_loader.dataset))) |

### 主函数与结果：

主函数：

|  |
| --- |
| 1. def main(): 2. device = torch.device('cuda') 3. transform = transforms.ToTensor() 4. datasets1 = datasets.MNIST('../data',train = True,download=True,transform=transform) 5. datasets2 = datasets.MNIST('../data',train = False,transform=transform) 6. train\_kws = {'batch\_size':64,'num\_workers':4,'pin\_memory':True,'shuffle':True} 7. test\_kws = {'batch\_size':1000,'num\_workers':2,'pin\_memory':True,'shuffle':True} 8. train\_loader = Data.DataLoader(datasets1,\*\*train\_kws) 9. test\_loader = Data.DataLoader(datasets2,\*\*test\_kws) 10. model = Net().to(device) 11. optimizer = optim.Adam(model.parameters(),lr = 0.001) 12. for epoch in range(10): 13. train(model,device,train\_loader,optimizer,epoch) 14. test(model,device,test\_loader) 15. torch.save(model,'net1.pt') |

结果如下图所示：



# 实验小结

在本次实验中，我使用了Keras基于MLP方法、PaddlePaddle基于LeNet方法和Pytorch基于CNN的方法完成，从结果来看Pytorch+CNN的效果最好，因为pytorch对GPU的支持很好，所以我设置了更多的训练轮次，达到了较好的结果。接下来，Paddle+LeNet结果更好一些，因为网络的结构更加适合这个模型；但是Keras+MLP的层数更少，训练的开销较小，且每轮训练的时间也较短，同时，结果也很不错。

对于这三个API，我感觉Keras的API使用更加的方便快捷；Pytorch对GPU加速的支持更好，训练的效率较高，但使用时的代码更加复杂；Paddle虽然现在的占比没有前两个那么高，但我相信在广大国内开发者的共同努力下或变得更好。

通过本次实验，我对使用神经网络进行数据挖掘有了更加直观的理解与认识，同时，通过对这三个主流API的分别尝试，我对这些深度学习API也有了更加直观的了解。总之，在这次实验中我收获很多，再次感谢江老师对实验的精心安排与对我们的认真教导。