



南開大學
Nankai University

计算机学院
机器学习实验报告

实验一 **Softmax Regression**

姓名：赵康明

学号：2110937

专业：计算机科学与技术

目录

1 实验目的	2
2 实验要求	2
3 实验原理	2
3.1 Softmax 函数	2
3.2 损失函数	3
3.3 梯度下降	3
3.4 正则化	3
4 实验代码	4
4.1 softmax-regression.py	4
4.2 evaluate.py	5
5 实验结果	5
6 实验改进和优化	6
6.1 小批次处理	6
6.1.1 小批次处理代码展示	6
6.1.2 实验结果	7
6.2 神经网络模型	8
6.2.1 实验原理	8
6.2.2 实验步骤	8
6.2.3 实验代码	9
6.2.4 实验结果	11
7 实验总结	11

1 实验目的

在这个练习中，需要训练一个分类器来完成对 MNIST 数据集中 0-9 的 10 个手写数字的分类，公开数据集可以在网站中找到，具体的实验框架也已经给出，需要根据代码框架理解训练流程并填充必要的代码完成对于手写体的识别工作。

2 实验要求

在本次实验中，我们将实现 softmax_regression 损失函数和梯度的计算：

目标函数 $J(\theta, x, y)$ softmax_regression 的损失函数表示为：

$$J(\theta, x, y) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K 1\{y^{(i)} = k\} \log \left(\frac{e^{\theta_k^T x^{(i)}}}{\sum_{j=1}^K e^{\theta_j^T x^{(i)}}} \right)$$

其中， m 是样本数， K 是类别数， θ 是参数向量， $x^{(i)}$ 是第 i 个样本的特征向量， $y^{(i)}$ 是第 i 个样本的真实类别， $1\{y^{(i)} = k\}$ 是指示函数， e 是自然对数底。

梯度 $\nabla_{\theta} J(\theta, x, y)$ 目标函数的梯度可以表示为：

$$\nabla_{\theta} J(\theta, x, y) = -\frac{1}{m} \sum_{i=1}^m (x^{(i)} (1\{y^{(i)} = k\} - P(y^{(i)} = k | x^{(i)}; \theta)))$$

其中， $P(y^{(i)} = k | x^{(i)}; \theta)$ 是样本 $x^{(i)}$ 属于类别 k 的概率，可以通过 softmax 函数计算。

3 实验原理

3.1 Softmax 函数

Softmax，主要用于多类别分类问题，将一组分数（通常称为 logits 或 scores）转化为概率分布，使每个类别的输出概率在 0 到 1 之间，并且所有类别的概率之和等于 1。Softmax 函数的输出通常用于确定输入数据属于哪个类别。

为了估计所有可能类别的条件概率，我们需要一个有多个输出的模型，每个类别对应一个输出。为了解决线性模型的分类问题，我们需要和输出一样多的仿射函数（affine function）。每个输出对应于它自己的仿射函数。以四个特征和 3 个可能的输出为例，我们有如下计算预测式子：

$$o_1 = x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1 \quad (1)$$

$$o_2 = x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2 \quad (2)$$

$$o_3 = x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3 \quad (3)$$

有了这样的输出之后，我们需要将这些输出通过 Softmax 函数映射到 $[0,1]$ 区间内，同时保证其非负性。而 Softmax 函数能够将未规范化的预测变换为非负数并且总和为 1，同时让模型保持可导的性质。为了完成这一目标，我们首先对每个未规范化的预测求幂，这样可以确保输出非负。为了确保最终输出的概率值总和为 1，我们再让每个求幂后的结果除以它们的总和。如下式：

$$\hat{y} = \text{softmax}(\mathbf{o}), \quad \text{其中 } \hat{y}_j = \frac{e^{o_j}}{\sum_k e^{o_k}}$$

在预测过程中, 由于对于所有的 j 总有 $0 \leq \hat{y}_j \leq 1$, 因此 \hat{y} 可以被视为一个正确的概率分布。Softmax 运算不会改变未规范化的预测 \mathbf{o} 之间的大小次序, 只会确定分配给每个类别的概率。因此, 我们仍然可以使用以下方式来选择最有可能的类别:

$$\arg \max_j \hat{y}_j = \arg \max_j o_j$$

尽管 softmax 是一个非线性函数, 但 softmax 回归的输出仍然由输入特征的仿射变换决定。因此, softmax 回归是一个线性模型。

3.2 损失函数

softmax 函数我们得到了一个向量 \hat{y} , 我们将使用交叉熵来定义其损失函数: 我们的损失函数可以表示为:

$$\text{Loss}(y, \hat{y}) = - \sum_{j=1}^q y_j \log \hat{y}_j$$

在这里, $L(y, \hat{y})$ 表示损失函数, y_j 和 \hat{y}_j 分别表示 y 和 \hat{y} 的第 j 个元素。 $\sum_{j=1}^q$ 表示对所有 j 的求和, $-\log$ 表示自然对数的负对数。

3.3 梯度下降

由损失函数展开可以得到我们完整的损失函数:

$$J(\theta, x, y) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K 1\{y^{(i)} = k\} \log \left(\frac{e^{\theta_k^T x^{(i)}}}{\sum_{j=1}^K e^{\theta_j^T x^{(i)}}} \right)$$

然后, 我们计算损失函数 $J(\theta, x, y)$ 关于 θ 的偏导数:

$$\nabla_{\theta} J(\theta, x, y) = -\frac{1}{m} \sum_{i=1}^m (x^{(i)} (1\{y^{(i)} = k\} - P(y^{(i)} = k|x^{(i)}; \theta)))$$

其中, $P(y^{(i)} = k|x^{(i)}; \theta)$ 是样本 $x^{(i)}$ 属于类别 k 的概率, 可以通过 softmax 函数计算。

3.4 正则化

为了防止模型过拟合, 我们考虑引入正则项来保证模型的拟合精度及正确性, 保证其拥有较好的鲁棒性, 而常用的正则化项有:

- L1 正则化: 也称为 L1 范数正则化, 它在损失函数中添加了模型参数的绝对值之和。L1 正则化的目标是促使模型参数稀疏化, 即让大部分参数为零。这对于特征选择和模型解释性很有用。

损失函数中的 L1 正则化项: $\lambda \sum_{i=1}^n |w_i|$

- L2 正则化: 也称为 L2 范数正则化, 它在损失函数中添加了模型参数的平方之和。L2 正则化的目标是防止模型参数过大, 从而减少过拟合。它对模型的影响通常比 L1 正则化更平滑。

损失函数中的 L2 正则化项: $\lambda \sum_{i=1}^n w_i^2$

在本次实验中我们考虑采用了 L1 正则化进行梯度下降的计算。

4 实验代码

4.1 softmax-regression.py

实验代码完成部分主要用到的库是 numpy 中的库函数中的矩阵乘法、求和和对数运算等函数，代码展示如下：

```
1 def softmax_regression(theta, x, y, iters, alpha):
2     # TODO: Do the softmax regression by computing the gradient and
3     # the objective function value of every iteration and update the theta
4     # 损失值f
5     f=list()
6     lam=0.00
7     data=x.T
8
9     for i in range (iters):
10         x=np.dot(theta,data) # x=权重*样本数据
11         exp_x=np.exp(x) # e^x
12         exp_x_sum=exp_x.sum(axis=0)
13
14         ## 交叉熵
15         y_hat=(exp_x/exp_x_sum) #y^
16
17         loss=0.0
18         log_y_hat=np.log(y_hat)
19         ## 计算损失函数
20         for j in range(y.shape[1]):
21             loss+=np.dot(log_y_hat[:,j],y_hat[:,j]) ## loss =xigemalogy *y
22
23         # 平均值
24         train_loss=-(1.0/y.shape[1])*loss
25         print("train_loss:",train_loss)
26         f.append(train_loss)
27
28         ## 计算梯度
29         batch_size=y.shape[1]
30         g=-(1.0/batch_size)*np.dot((y-y_hat),data.T)+lam*theta
31         theta=theta-alpha*g
32     return theta
```

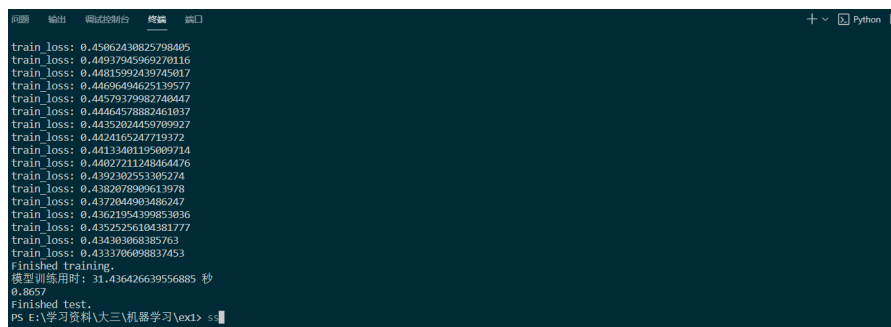
4.2 evaluate.py

准确率评估函数完善如下：

```
1 def softmax_regression(theta, x, y, iters, alpha):
2 def cal_accuracy(y_pred, y):
3     # TODO: Compute the accuracy among the test set and store it in acc
4     correct=0
5     for i in range(len(y)):
6         if(y_pred[i]==y[i]):
7             correct+=1
8     acc=correct/len(y)*1.0
9     return acc
```

5 实验结果

我们设置了迭代次数为 100，得到实验结果如下：



```
train_loss: 0.45062438825798405
train_loss: 0.44937945969270116
train_loss: 0.44815992439745017
train_loss: 0.44696494625139577
train_loss: 0.44579379982740447
train_loss: 0.44464578882461037
train_loss: 0.44352024459709927
train_loss: 0.4424165247719372
train_loss: 0.44133401195009714
train_loss: 0.44027211248464476
train_loss: 0.4392302553305274
train_loss: 0.4382078999613978
train_loss: 0.4372044303486247
train_loss: 0.43621954399853036
train_loss: 0.43525256104381777
train_loss: 0.434303068385763
train_loss: 0.4333706998837453
Finished training.
模型训练用时: 31.436426639556885 秒
0.8657
Finished test.
PS E:\学习资料\大三\机器学习\ex1> ss
```

图 5.1: 实验结果

可以看到经过 100 次迭代训练后，我们得到了 86.57% 的正确率，正确率较高。

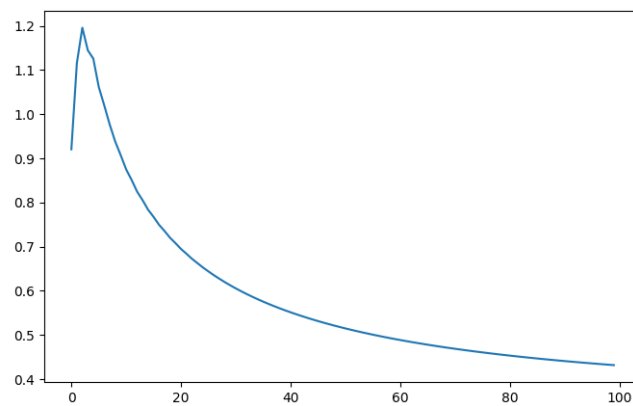


图 5.2: 损失率

而从损失图我们也可以看出，随着训练次数的增加，损失率越来越低，比较符合我们的预期。

6 实验改进和优化

6.1 小批次处理

在计算梯度时候采用的是 for 循环进行梯度计算，效率较低，于是考虑通过小批次处理计算梯度的方式提高效率。即在计算梯度的时候，将数据按批次划分进行处理计算。其有以下优势：


1. 小批次处理可以帮助模型更好地泛化到新数据。每个小批次都是从整个数据集中随机选择的，这有助于模型避免陷入局部最小值，并提高泛化性能。
2. 模型收敛：小批次处理通常使模型更快地收敛到损失函数的局部最小值，因为每个小批次都可以更新模型参数，从而加速收敛过程。
3. 稳定性：小批次处理可以提高训练的稳定性。在某些情况下，随机梯度下降可能导致参数更新的不稳定性，小批次处理可以减轻这种问题

6.1.1 小批次处理代码展示

```
1
2 def mini_batch_gradient_descent(theta, x, y, iters, alpha, batch_size, lam=0.001):
3     # 的形状是 (60000, 784), y 的形状是 (10, 60000)
4
5     # 定义超参数
6     batch_size = 64 # 小批次大小
7     num_samples = x.shape[0] # 样本总数
8     num_batches = num_samples // batch_size # 总批次数
9
10    # 随机初始化模型参数 theta, 假设 theta 的形状是 (10, 784)
11    theta = np.random.rand(10, 784)
12
13    # 定义学习率和迭代次数等超参数
14    learning_rate = 0.01
15    num_epochs = 100
16    # 迭代训练
17    for epoch in range(num_epochs):
18        # 随机打乱数据集的顺序
19        permutation = np.random.permutation(num_samples)
20        x_shuffled = x[permutation]
21        y_shuffled = y[:, permutation]
22        for batch in range(num_batches):
23            # 获取当前小批次数据
24            start = batch * batch_size
```

```
25     end = (batch + 1) * batch_size
26     x_batch = x_shuffled[start:end]
27     y_batch = y_shuffled[:, start:end]
28     # 执行前向传播, 计算损失, 计算梯度
29     logits = np.dot(theta, x_batch.T)
30     exp_x = np.exp(logits)
31     exp_x_sum = np.sum(exp_x, axis=0)
32     y_hat = exp_x / exp_x_sum
33     loss = -np.sum(y_batch * np.log(y_hat)) / batch_size
34     gradient = -(1.0 / batch_size) * np.dot((y_batch - y_hat), x_batch)
35         + lam*theta # 计算梯度
36     # 更新模型参数
37     theta -= learning_rate * gradient
38     # 打印损失或其他训练过程中的指标
39     print(f"Epoch {epoch+1}, Loss: {loss}")
40
41 return theta
```

6.1.2 实验结果



```
Epoch 83, Loss: 0.36071345343993744
Epoch 84, Loss: 0.19024560239123411
Epoch 85, Loss: 0.2854845260548502
Epoch 86, Loss: 0.27431302205312336
Epoch 87, Loss: 0.5662099880258671
Epoch 88, Loss: 0.6704412854172455
Epoch 89, Loss: 0.1254341539363219
Epoch 90, Loss: 0.21564314902133044
Epoch 91, Loss: 0.2926761060859186
Epoch 92, Loss: 0.517203927700527
Epoch 93, Loss: 0.24270632243095658
Epoch 94, Loss: 0.17663032616648608
Epoch 95, Loss: 0.32451334068040116
Epoch 96, Loss: 0.14988748272693392
Epoch 97, Loss: 0.08332137415089708
Epoch 98, Loss: 0.2893615171648941
Epoch 99, Loss: 0.25903168971587714
Epoch 100, Loss: 0.35033018227606527
Finished training.
模型训练用时: 36.64919137954712 秒
0.9165
Finished test.
```

图 6.3: 小批次处理实验结果

可以发现我们的正确率由原来的 86.57% 提升到了 91.65%。正确率得到了明显的提高。

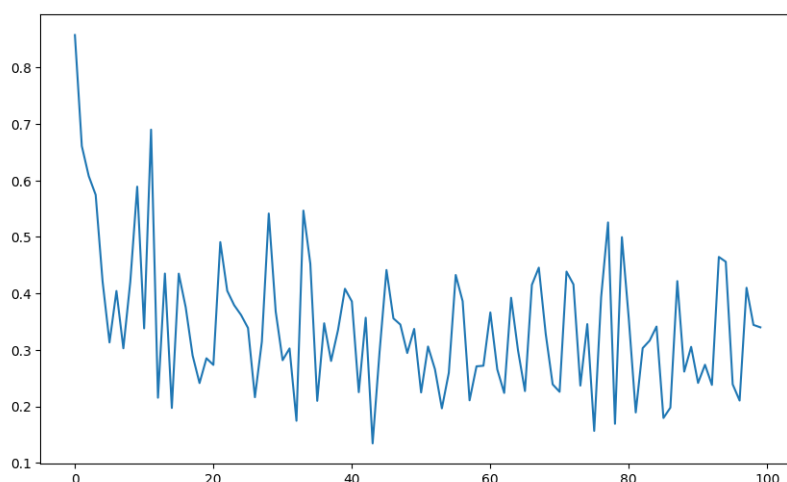


图 6.4: 小批次损失率

我们可以发现小批次的模型训练过程中，损失率波动的很厉害，可能原因是超参数的选取不够正确，或者是数据分布的原因，由于其准确率较高，不在此对损失率如此波动进行进一步探讨。

6.2 神经网络模型

6.2.1 实验原理

1. 神经网络架构在本实验中，我们使用了一个简单的全连接神经网络模型。该模型包括多个线性层和激活函数，用于对 MNIST 手写数字进行分类。神经网络的架构如下：
 - (a) 输入层：Flatten 层，将 28×28 的图像展平成 784 个节点。
 - (b) 隐藏层：多个具有 ReLU 激活函数的线性层，每一层的节点数逐渐减少。
 - (c) 输出层：具有 Softmax 激活函数的线性层，用于进行多类别分类。
2. 数据预处理在数据预处理阶段，我们使用了 `torchvision.transforms` 模块中的 `ToTensor` 和 `Normalize` 转换。`ToTensor` 将图像转换为 PyTorch 张量，并将其值缩放到 0 到 1 的范围。`Normalize` 对张量进行归一化，使其均值为 0.5，标准差为 0.5。
3. 损失函数和优化器我们使用了交叉熵损失函数（`CrossEntropyLoss`）来衡量模型输出与真实标签之间的差异。优化器选择了 Adam，并设置了不同的学习率进行比较。
4. 实验设计我们设计了三个实验，分别使用学习率为 0.001、0.0005 和 0.0001，比较它们在相同训练周期内在测试集上的损失和准确率。

6.2.2 实验步骤

1. 模型定义：定义全连接神经网络模型。
2. 数据加载与预处理：下载 MNIST 数据集并进行训练集和测试集的加载，并对图像进行归一化处理。

3. 训练过程：使用不同的学习率对模型进行训练，记录训练过程中的损失和准确率。
4. 实验结果分析：比较不同学习率下的测试损失和测试准确率曲线

6.2.3 实验代码

1. 神经网络的搭建

```
1     import torch
2     import torchvision
3     from tqdm import tqdm
4     from matplotlib import pyplot
5
6     # %%
7     # 定义全连接网络模型
8     class Net(torch.nn.Module):
9         def __init__(self):
10             super(Net, self).__init__()
11             self.model = torch.nn.Sequential(
12                 torch.nn.Flatten(),
13                 torch.nn.Linear(in_features=784, out_features=512),
14                 torch.nn.ReLU(),
15                 torch.nn.Linear(in_features=512, out_features=256),
16                 torch.nn.ReLU(),
17                 torch.nn.Linear(in_features=256, out_features=128),
18                 torch.nn.ReLU(),
19                 torch.nn.Linear(in_features=128, out_features=64),
20                 torch.nn.ReLU(),
21                 torch.nn.Linear(in_features=64, out_features=32),
22                 torch.nn.ReLU(),
23                 torch.nn.Linear(in_features=32, out_features=10),
24                 torch.nn.Softmax(dim=1)
25             )
26         def forward(self, input):
27             output = self.model(input)
28             return output
29     # %%
30     device = "cuda:0" if torch.cuda.is_available() else "cpu" #
31     # 检测并选择GPU或CPU设备
32     transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
33                                                 torchvision.transforms.Normalize(mean=[0.5],
34                                                                                   std=[0.5])]) # 数据预处理：归一化
33
34     BATCH_SIZE = 128 # 批量大小
```

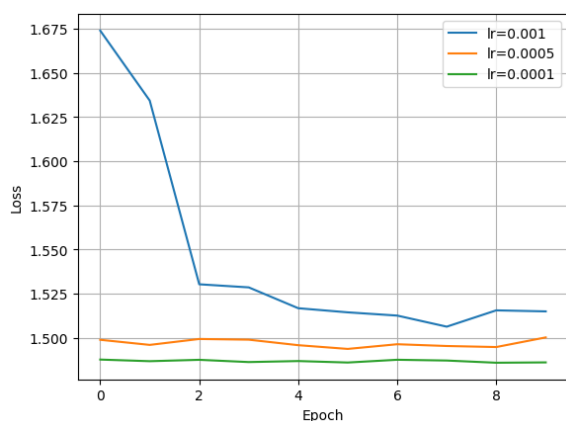
```
35 EPOCHS = 10    # 迭代次数
36 lossF = torch.nn.CrossEntropyLoss() # 多分类问题的损失函数为交叉熵
```

2. 训练过程的实现

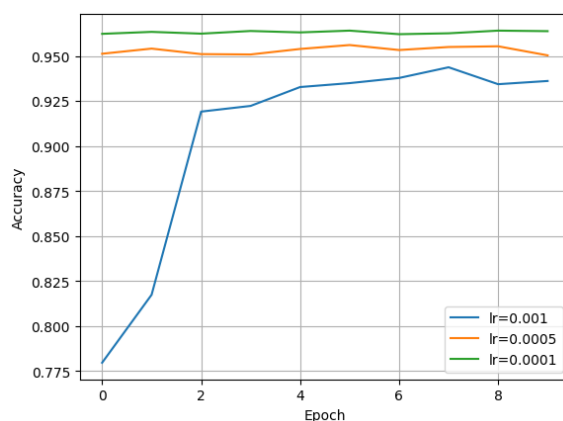
```
1 net = Net()
2 optimizer = torch.optim.Adam(net.parameters(), lr=0.001) #
   优化器为Adam, 学习率为0.001
3 history_0001 = {'Test Loss': [], 'Test Accuracy': []}
4 for epoch in range(1, EPOCHS + 1):
5     progressBar = tqdm(trainDataLoader, unit='step')
6     net.train(True)
7     for step, (train_imgs, labels) in enumerate(progressBar):
8         train_imgs = train_imgs.to(device)
9         labels = labels.to(device)
10
11         net.zero_grad()
12         outputs = net(train_imgs)
13         loss = lossF(outputs, labels) # 计算损失
14         predictions = torch.argmax(outputs, dim=1) # 进行预测
15         accuracy = torch.sum(predictions == labels) / labels.shape[0] #
           测试准确率
16         loss.backward() # 反向传播
17
18         optimizer.step()
19         progressBar.set_description("[%d/%d] Loss: %.4f, Acc: %.4f" %
20                                     (epoch, EPOCHS, loss.item(), accuracy.item()))
21     if step == len(progressBar) - 1:
22         correct, totalLoss = 0, 0
23         net.train(False)
24         for test_imgs, labels in testDataLoader:
25             test_imgs = test_imgs.to(device)
26             labels = labels.to(device)
27             outputs = net(test_imgs)
28             loss = lossF(outputs, labels)
29             predictions = torch.argmax(outputs, dim=1)
30             totalLoss += loss
31             correct += torch.sum(predictions == labels)
32         testAccuracy = correct / (BATCH_SIZE * len(testDataLoader)) #
           测试准确率
33         testLoss = totalLoss / len(testDataLoader) # 测试损失
34         history_0001['Test Loss'].append(testLoss.item())
```

```
35     history_0001['Test Accuracy'].append(testAccuracy.item())
36     progressBar.set_description("[%d/%d] Loss: %.4f, Acc: %.4f, Test
37                               Loss: %.4f, Test Acc: %.4f" %
38                               (epoch, EPOCHS, loss.item(), accuracy.item(),
39                               testLoss.item(),
40                               testAccuracy.item()))
41     progressBar.close()
```

6.2.4 实验结果



(a) 测试图像



(b) 灰度图

图 6.5

测试损失曲线比较：随着学习率的减小，测试损失在初始阶段可能下降得更为缓慢，但可能在后续阶段达到更低的损失值。我们发现 0.0001 的学习率损失值一直都较小，且经过 10 轮学习之后，其损失值最小。

测试精确率比较：随着学习率的减小，测试的精确值一直处于较高的状态，最高得到了 **96.38%** 的正确率。

这是因为更小的学习率可以更精细地搜索参数空间，较小的学习率可以使模型更小心地调整参数，有助于在参数空间中更细致地搜索，从而找到更好的局部极小值。

7 实验总结

本次实验利用了 numpy 库所提供的函数完成基础的 softmax 回归的手搓实现，在训练 100 次的情况下得到了 **86%** 的正确率，在此基础上还使用了小批次训练迭代的方式，使得正确率提高到了 **91%** 左右。在此基础上，额外探索了神经网络模型对于手写数字识别，建立简单的全连接神经网络，并对比不同学习率下的所训练出的模型的准确率和损失值，得到了 **96.38%** 的准确率。