神经网络实验报告

实验目的：

对收集的Mushroom数据进行训练和测试，使其能够根据特征来预测最终的class标签，是有毒的（**p**oisonous）或是可食用的（**e**dible）。

数据集包括8124行以及23列，其中特征共22种。

```
import pandas as pd
mushrooms= pd.read_csv('D:\学习相关\专业英语\datasets_478_974_mushrooms.csv')
mushrooms.head()
```

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil-type	veil-color	ring-number	ring-type	spore-print-color	population
0	p	x	s	n	t	p	f	c	n	k	...	s	w	w	p	w	o	p	k	s
1	e	x	s	y	t	a	f	c	b	k	...	s	w	w	p	w	o	p	n	n
2	e	b	s	w	t	l	f	c	b	n	...	s	w	w	p	w	o	p	n	n
3	p	x	y	w	t	p	f	c	n	n	...	s	w	w	p	w	o	p	k	s
4	e	x	s	g	f	n	f	w	b	k	...	s	w	w	p	w	o	e	n	a

5 rows × 23 columns

实验过程：

一.数据预处理

1.通过调用 load_data 模块的load_data()方法，将原始数据调整为数字形式，返回特征输入和标签输出。

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder

def load_data():

    mushrooms= pd.read_csv('D:\学习相关\专业英语\datasets_478_974_mushrooms.csv')
    X=mushrooms.drop('class',axis=1) #去掉class这一列，得到输入的22个特征
    y=mushrooms['class'] #得到class标签
    Encoder_X = LabelEncoder() #字母转换为数字
    for col in X.columns:
        X[col] = Encoder_X.fit_transform(X[col])
    X=np.array(X)

    Encoder_y=LabelEncoder()
    y = Encoder_y.fit_transform(y)
    return X,y
```

2.调用load_data模块中的adjust_data()方法，将数据集以7:3的比例分为训练集、测试集，调整数据格式，使得数据变成元组形式，输入层有22个神经元，输出层有2个神经元。

```python
def vectorized_result(j):
    e = np.zeros((2, 1))
    e[j] = 1.0
    return e

#调整格式，使得数据变成元组形式，输入层有22个神经元，输出层有2个神经元
def adjust_data(X,y):
    X_train, X_test, y_train, y_test = train_test_split(X, y, 
test_size=0.3,random_state=0 )#分类
    #对训练集调整

    train_in = np.array([x[0:] for x in X_train]).astype('float')#转换成浮点类型
    train_vect=[np.reshape(x, (22,1)) for x in train_in]#变成向量形式
    train_out = [vectorized_result(y) for y in y_train]

    #测试集同理
    test_in=np.array([x[0:] for x in X_test]).astype('float')
    test_vect=[np.reshape(x, (22,1)) for x in test_in]
    test_out = [vectorized_result(y) for y in y_test]

    train_datasets = list(zip(train_vect, train_out))#形成输入层、输出层的元组
    test_datasets = list(zip(test_vect, test_out))

    return train_datasets,test_datasets
```

二.神经网络的搭建

1.导入相关模块

```python
import pandas as pd
import numpy as np
import random
import json
import sys
```

2.创建代价函数的2个类，分别是交叉熵代价函数以及二次代价函数。（用于后续比较）

方法重写的目的：

- fn 方法用来衡量输出激活值a 和目标输出 y 差距优劣的度量。
- delta 方法用于计算在反向传播时的网络输出误差 δ^L ，且不同的代价函数，输出误差的形式就不同。

```python
class CrossEntropyCost(object):#交叉熵代价函数
    @staticmethod
    def fn(a, y):
        return np.sum(np.nan_to_num(-y*np.log(a)-(1-y)*np.log(1-a)))
    #np.nan_to_num 调用确保了 Numpy 正确处理接近 0 的对数值
    @staticmethod
    def delta(z, a, y):
        return (a-y)
```

```python
class QuadraticCost(object):#二次代价函数
    @staticmethod
    def fn(a, y):
        return 0.5*np.linalg.norm(a-y)**2
    @staticmethod
    def delta(z, a, y):
        return (a-y) * sigmoid_prime(z)
```

3.创建本实验中用到的激活函数 $sigmoid$ 函数以及它的导数。

```python
#本实验中用到的激活函数，sigmoid函数以及它的导数
def sigmoid(x):
    return 1/(1+np.exp(-x))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

4.（关键）准备好前期工作以后，接下来是对神经网络的初始化，创建神经网络类。

- **4.1** 初始化神经网络层数，设置权重和偏置的初始值。

```python
#初始化神经网络
'''-----------------------------Network类-------------------------------'''
class Network(object):

    def __init__(self, sizes, cost):

        self.num_layers = len(sizes)
        self.sizes = sizes
        self.large_weight_initializer()
        self.cost=cost
```

- **4.2** 权重、偏置初始化，large_weight_initializer 使用了均值为0而标准差为1的高斯分布。

```python
    def large_weight_initializer(self):

        self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(self.sizes[:-1], self.sizes[1:])]
```

- **4.3** 前馈传播，代入输入值，信号从输入层向输出层单向传播。

```
    def feedforward(self, a):

        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a
```

- **4.4**（关键点）随机梯度下降算法，将训练集分为一个个mini_batch_size大小的小批量数据，对每一个小批量数据调用update_mini_batch（）方法来更新权重和偏置。
  - 在update_mini_batch（）中调用backproc （）方法计算偏导数。
  - 更新权重用到了L2正则化计算式子，即 $self.weights = [1 - eta * ...]$ 这一行

    对应的数学公式为：

$$w = \left(1 - \frac{\eta\lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}$$

```
    def SGD(self, training_data, epochs, mini_batch_size, eta,
            lmbda = 0.0,
            evaluation_data=None,
            monitor_evaluation_cost=False,
            monitor_evaluation_accuracy=False,
            monitor_training_cost=False,
            monitor_training_accuracy=False):

        if evaluation_data: #evaluation_data=test_data
            n_data = len(evaluation_data)#number of test_data
        n = len(training_data)
        evaluation_cost, evaluation_accuracy = [], []#测试集的代价和准确率
        training_cost, training_accuracy = [], []#训练集的代价和准确率
        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]#将训练分为一个个的mini_batch
            for mini_batch in mini_batches:
                self.update_mini_batch(
                    mini_batch, eta, lmbda, len(training_data))#对每一个
mini_batch更新权重和偏置
            print ("Epoch %s training complete" % j)
            if monitor_training_cost:
                cost = self.total_cost(training_data, lmbda)
                training_cost.append(cost)
                print ("Cost on training data: {}".format(cost))
            if monitor_training_accuracy:
                accuracy = self.accuracy(training_data, convert=True)
                training_accuracy.append(accuracy)
                print ("Accuracy on training data: {} / {}".format(
                    accuracy, n))
            if monitor_evaluation_cost:
                cost = self.total_cost(evaluation_data, lmbda, convert=True)
                evaluation_cost.append(cost)
                print ("Cost on evaluation data: {}".format(cost))
            if monitor_evaluation_accuracy:
                accuracy = self.accuracy(evaluation_data)
                evaluation_accuracy.append(accuracy)
                print ("Accuracy on evaluation data: {} / {}".format(
                    self.accuracy(evaluation_data), n_data))
```

```
        return evaluation_cost, evaluation_accuracy, \
        training_cost, training_accuracy


def update_mini_batch(self, mini_batch, eta, lmbda, n):

    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)#反向传播
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [(1-eta*(lmbda/n))*w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]


def backprop(self, x, y):

    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
    delta = (self.cost).delta(zs[-1], activations[-1], y)
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())

    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
```

- **4.5** 计算准确率、代价（在SGD方法中调用）。

```
def accuracy(self, data,convert=True):

    results = [(np.argmax(self.feedforward(x)), np.argmax(y))
                for (x, y) in data]
    #用于观察测试后的 向量中最大值的索引 与 真实值的最大值的索引位置 是否相同
    return sum(int(x == y) for (x, y) in results)
```

```
    def total_cost(self, data, lmbda, convert=False):

        cost = 0.0
        for x, y in data:
            a = self.feedforward(x)
            if convert: y = vectorized_result(y)
            cost += self.cost.fn(a, y)/len(data)
        cost += 0.5*(lmbda/len(data))*sum(
            np.linalg.norm(w)**2 for w in self.weights)
        return cost
```

- **4.6** 保存模型为json格式

```
    def save(self, filename):

        data = {"sizes": self.sizes,
                "weights": [w.tolist() for w in self.weights],
                "biases": [b.tolist() for b in self.biases],
                "cost": str(self.cost.__name__)}
        f = open(filename, "w")
        json.dump(data, f)
        f.close()

'''-------------------------Network类结束------------------------------'''
```

5.用于加载json数据，方便以后直接调用。

```
#用于加载保存好后的json数据
def load(filename):
    """Load a neural network from the file ``filename``.  Returns an
    instance of Network.
    """
    f = open(filename, "r")
    data = json.load(f)
    f.close()
    cost = getattr(sys.modules[__name__], data["cost"])
    net = Network(data["sizes"], cost=cost)
    net.weights = [np.array(w) for w in data["weights"]]
    net.biases = [np.array(b) for b in data["biases"]]
    return net
```

三.数据可视化

1.在主方法中加载数据

```
import network
import load_data
import make_plot
import numpy as np
import matplotlib.pyplot as plt
import imp
X,y=load_data.load_data()#加载输入和输出
```

```
X
```

```
array([[5, 2, 4, ..., 2, 3, 5],
       [5, 2, 9, ..., 3, 2, 1],
       [0, 2, 8, ..., 3, 2, 3],
       ...,
       [2, 2, 4, ..., 0, 1, 2],
       [3, 3, 4, ..., 7, 4, 2],
       [5, 2, 4, ..., 4, 1, 2]])
```

```
y
```

```
array([1, 0, 0, ..., 0, 1, 0])
```

- 分类成训练集和测试集

```
train_data,test_data=load_data.adjust_data(X,y)    #分类：训练集、测试集
```

2.通过控制变量的方法，绘制图表，得出最佳参数。

- **2.1** 测试相同参数下，二次代价函数和交叉熵代价函数的准确率

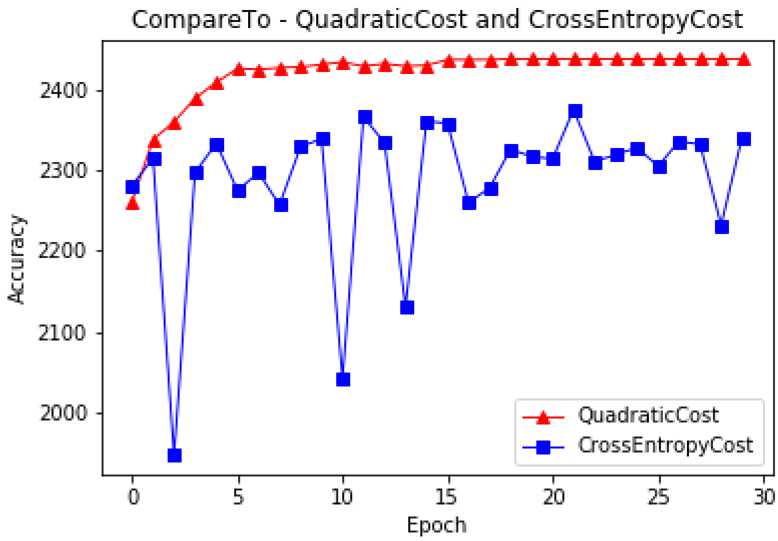
  ```
  make_plot.compare_cost(train_data,test_data)
  ```

  ```
  def compare_cost(train_data,test_data):
      layers = [22,50,2]
      epochs = 30
      mini_batch = 10
      eta = 0.5
      #lmbda默认是0.0
      net1 = network.Network(layers, cost=network.QuadraticCost)#二次代价函数
      accuracy1=net1.SGD(train_data, epochs, mini_batch, eta, evaluation_data
  = test_data, \
      monitor_evaluation_accuracy = True)

      net2 = network.Network(layers, cost=network.CrossEntropyCost)#交叉熵代价函
  数
      accuracy2=net2.SGD(train_data, epochs, mini_batch, eta, evaluation_data
  = test_data, \
      monitor_evaluation_accuracy = True)

      x=np.arange(0,epochs)
  ```

```
    plt.plot(x, accuracy1[1], color="r", linestyle="-",
marker="^",label="QuadraticCost", linewidth=1)
    plt.plot(x, accuracy2[1], color="b", linestyle="-",
marker="s",label="CrossEntropyCost", linewidth=1)
    plt.legend(loc='lower right')
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.title("CompareTo - QuadraticCost and CrossEntropyCost")
    plt.savefig("QC-CEC.png")
    plt.show()
```



可以得出二次代价函数随着迭代期增加，准确率趋于平稳，且高于交叉熵代价函数。

- **2.2** 控制其他参数不变，改变神经网络层数和神经元数，绘制二次代价函数和交叉熵代价函数的折线图

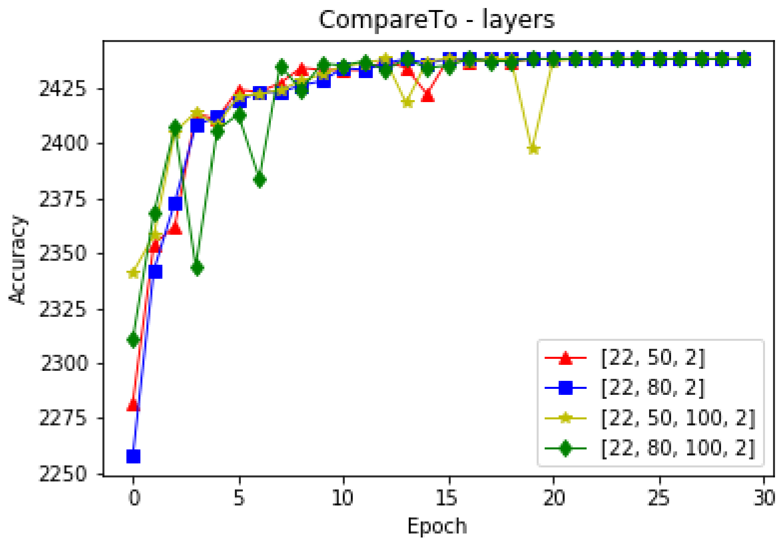
```
make_plot.compare_QC_layers(train_data,test_data)#二次代价函数
```

```
make_plot.compare_CEC_layers(train_data,test_data)#交叉熵代价函数
```

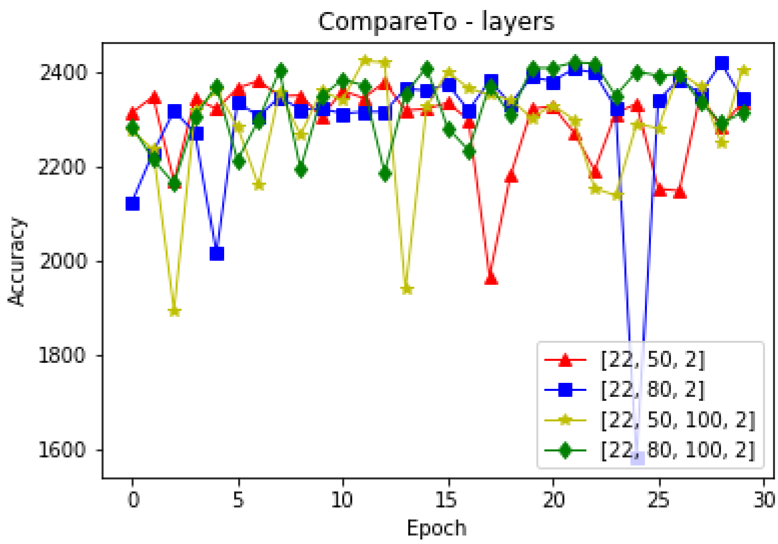
```
def compare_QC_layers(train_data,test_data):
    epochs=30
    mini_batch=10
    eta = 0.5
    lmbda = 5

    layers1=[22,50,2]
    layers2 = [22,80,2]
    layers3=[22,50,100,2]
    layers4=[22,80,100,2]
    '''  。。。后面省略 可在上传的代码中看。。。。 '''
```

二次代价函数：



交叉熵代价函数：

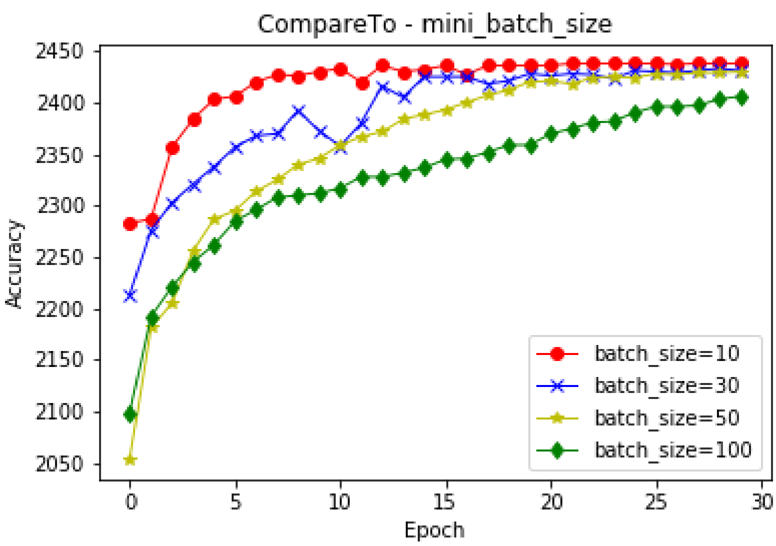


*根据上图可以得出二次代价函数随着层数增加、神经元增加，准确率的变化较为平缓，且高于交叉熵代价函数，**因此后面均使用二次代价函数**，设定层数大小为 [22,50,2]*

- **2.3** 控制其他参数不变，改变mini_batch的尺寸

```
make_plot.compare_mini_batch_size(train_data,test_data)
```

```
def compare_mini_batch_size(train_data,test_data):
    layers=[22,50,2]
    epochs=30
    eta = 0.5
    lmbda = 5
    mini_batch_size1=10
    mini_batch_size2=30
    mini_batch_size3=50
    mini_batch_size4=100
     '''  。。。后面省略 可在上传的代码中看。。。。 '''
```



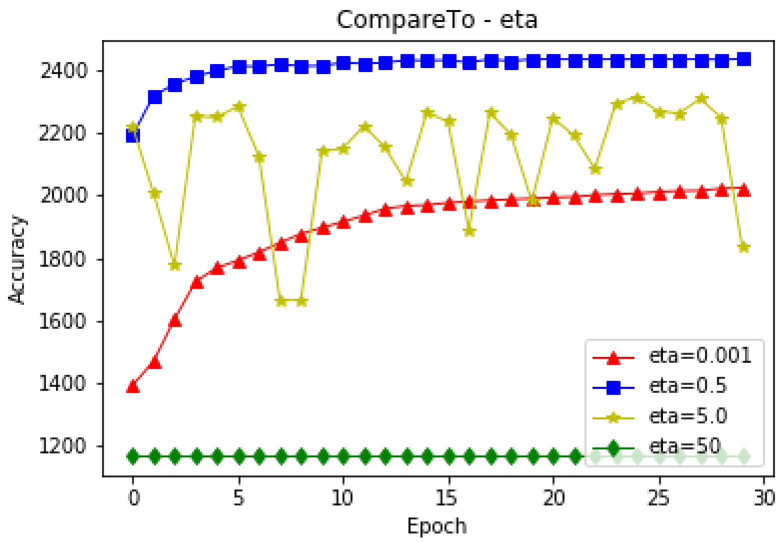
根据上图可以得出当batch_size=10时有较好的准确率

- **2.4** 控制其他参数不变，改变eta大小

```
make_plot.compare_eta(train_data,test_data)
```

```
def compare_eta(train_data,test_data):
    layers=[22,50,2]
    epochs=30
    lmbda = 5
    mini_batch=10

    eta1=0.001
    eta2=0.5
    eta3=5.0
    eta4=50
     '''  。。。后面省略 可在上传的代码中看。。。。 '''
```



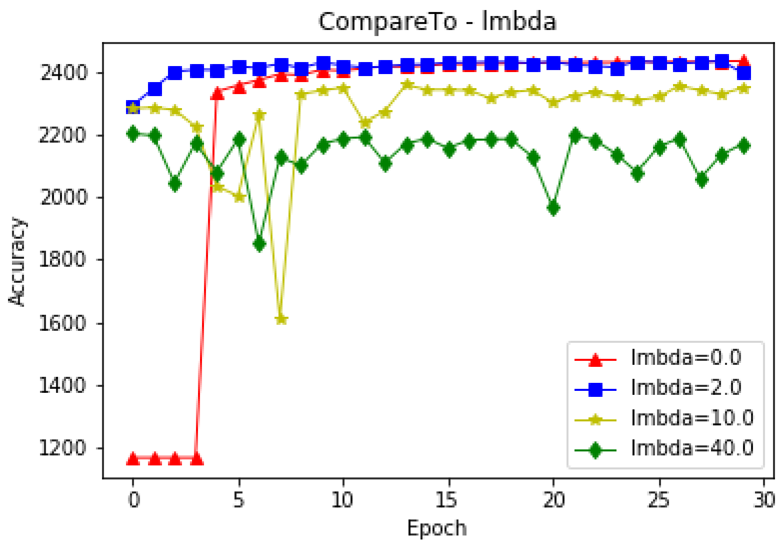
根据上图可以得出当eta=0.5时有较好的准确率

- **2.5** 控制其他参数不变，改变lmbda大小

```
make_plot.compare_lmbda(train_data,test_data)
```

```
def compare_lmbda(train_data,test_data):
    layers=[22,50,2]
    epochs=30
    eta=0.5
    mini_batch=10

    lmbda1=0.0
    lmbda2=2.0
    lmbda3=10.0
    lmbda4=40.0
     '''  。。。后面省略 可在上传的代码中看。。。。 '''
```



根据上图可以得出在lmbda=0.0 , 2.0时准确率较高（我两个都有测试过，后面发现lamba=0的时候正确率比较高，可能是对于数据量较小的数据，规范化参数不需要很大。）

因此，最终设定的参数为:

```
def best_network(train_data,test_data):
    layers=[22,50,2]
    epochs=30
    eta=0.5
    mini_batch=10
    lmbda=0.0

    net1 = network.Network(layers, cost=network.QuadraticCost)
    accuracy1=net1.SGD(train_data, epochs, mini_batch, eta,lmbda=lmbda1,
evaluation_data = test_data, \
    monitor_evaluation_accuracy = True)
```

最终得到的准确率为:

```
....
Epoch 19 training complete
Accuracy on evaluation data: 2436 / 2438
Epoch 20 training complete
Accuracy on evaluation data: 2433 / 2438
Epoch 21 training complete
Accuracy on evaluation data: 2437 / 2438
Epoch 22 training complete
Accuracy on evaluation data: 2437 / 2438
Epoch 23 training complete
Accuracy on evaluation data: 2437 / 2438
Epoch 24 training complete
Accuracy on evaluation data: 2438 / 2438
Epoch 25 training complete
Accuracy on evaluation data: 2437 / 2438
Epoch 26 training complete
Accuracy on evaluation data: 2438 / 2438
Epoch 27 training complete
Accuracy on evaluation data: 2438 / 2438
Epoch 28 training complete
Accuracy on evaluation data: 2437 / 2438
Epoch 29 training complete
Accuracy on evaluation data: 2438 / 2438
```

接近100%

实验结论

通过该实验，对构建神经网络有了深入的了解，其中，随机梯度下降是重点部分，之后再对各个参数进行测试、优化，得到最优神经网络。

参考资料：《Neural Network and Deep Learning》