手写数字分类的简单神经网络 (Python)

我们将把 MATLAB 面向对象的程序逐字逐句翻译成 Python 代码。Michael Nielsen 的 "Neural Networks and Deep Learning "一书中给出了 Python 程序，那里的运算是针对样本数据 x 循环处理的，而前面的 MATLAB 代码是一次性处理所有 x。该书的[英文电子版]()可在线阅读，百度上也可找到中译电子书。

数据加载

我们使用 Michael Nielsen 源码中给出的数据集，并把它转化为 MATLAB 代码的数据形式。下载的数据集为压缩包 mnist.pkl.gz ， 并保存在主目录下。数据加载函数如下

```
#### Libraries
# Standard library
import pickle
import gzip
# Third-party libraries
import numpy as np

# 加载 MINIST 数据集
def load_data():
    f = gzip.open('mnist.pkl.gz', 'rb')
    training_data, validation_data, test_data = pickle.load(f, encoding="latin1")
    f.close()
```

数据是以元组形式返回的，包含三部分：

- 训练数据 training_data：该数据也是一个元组，分别记录 50,000 个手写数据的图像信息和相应的数字 （0 到 9）。这 50,000 个数据是用 NumPy 的 ndarray 存储的，共有 50,000 行 784 列，每行对应一个手写数字的像素信息。
- 验证数据 validation_data
- 测试数据 test_data

后两个数据与训练数据形式一样，只不过都只有 10,000 个手写数字。

如下转化为 MATLAB 代码的数据形式

```
#### Libraries
# Standard library
import pickle
import gzip

# Third-party libraries
import numpy as np

def load_data():
    f = gzip.open('mnist.pkl.gz', 'rb')
    tr_d, va_d, te_d = pickle.load(f, encoding="latin1")
    f.close()
```

```python

    training_data_x = tr_d[0].transpose()
    training_data_y = np.hstack([vectorize_num(i) for i in tr_d[1]])
    validation_data_x = va_d[0].transpose()
    validation_data_y = np.hstack([vectorize_num(i) for i in va_d[1]])
    test_data_x = te_d[0].transpose()
    test_data_y = np.hstack([vectorize_num(i) for i in te_d[1]])

    return training_data_x, training_data_y, \
           validation_data_x, validation_data_y, \
           test_data_x, test_data_y

def vectorize_num(j):
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```

这里的 vectorize_num 就是把 0 到 9 的数字变为 10 维的二进制向量。

建立类的框架

对 MATLAB 程序，建立的类的框架如下

```matlab
classdef Network < handle
    properties
        num_layers
        sizes
        biases
        weights
    end

    methods
        % constructor: initialization
        function obj = Network(sizes)
            obj.num_layers = length(sizes);
            obj.sizes = sizes;
            for s = 1:obj.num_layers-1 % 2,...,L
                obj.biases{s} = randn(sizes(s+1),1);
                obj.weights{s} = randn(sizes([s+1,s]));
            end
        end

        % feedforward
        [aL,a,z] = feedforward(obj,data_x);

        % backpropagation
        backprop(obj,mini_batch_y,a,z,eta);

        % train network by SGD
        SGD(obj, training_data_x, training_data_y, epochs, mini_batch_size, eta, varargin);

        % evaluation of test_data
        [np,yp,y] = evaluate(obj,data_x,data_y);

```

```
    end  % end of methods

end
```

- 为了看到类的框架，我们把类方法的实现放在类外实现，其实可以直接放到类的内部实现。
- MATLAB 的函数名与文件名相同，类外实现比较容易。Python 中一般放在内部实现，这由其文件组织方式决定的。

Python 类的框架如下

```python
class Network(object):

    # constructor: initialization
    def __init__(self, ndim):
        self.num_layers = len(ndim)
        self.ndim = ndim
        self.biases = [np.random.randn(s, 1) for s in ndim[1:]]
        self.weights = [np.random.randn(s, t)
                        for s,t in zip(ndim[1:], ndim[:-1])]

    # feedforward
    def feedforward(self, data_x):
        ...

    # backpropagation
    def backprop(self,mini_batch_y,a,z,eta):
        ...

    # train network by SGD
    def SGD(self, training_data_x, training_data_y, epochs, mini_batch_size, \
             eta, test_data_x=None, test_data_y=None):
        ...

    # evaluation of test_data
    def evaluate(self, data_x, data_y):
        ...
```

编写类的方法

神经网络的初始化

神经网络如下建立

```python
# ---------- Create network ----------
import network
net = network.Network([784, 15, 10])
```

Network 对象如下初始化:

```python
#### Libraries
# Standard library
import random

```

```python
# Third-party libraries
import numpy as np
import matplotlib.pyplot as plt

class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(s, 1) for s in sizes[1:]]
        self.weights = [np.random.randn(s, t)
                        for s,t in zip(sizes[1:], sizes[:-1])]
```

这里,

- `sizes[1:]` 是切片的省略写法，完整写法为 `sizes[1:3:1]` 表示起点索引为 1，终点索引为 3（注意 Python 提取元素时不包含最后一个），步长为 1。这样，`size[1:] = [15,10]`。
- `sizes[:-1]` 的完整写法为 `sizes[0:-1:1]`，这里的起点索引为 0，终点索引为 -1 (也就是倒数第 1 个, 不包含该索引)，从而它的结果为 `[784, 15]`. 没有倒数第 0 个的说法，因为 -0 和 0 无法区别。
- 可以看到，Python 的循环要比 MATLAB 方便。

feedforward 方法

MATLAB 代码如下

```matlab
function [aL,a,z] = feedforward(obj,data_x)
    a = cell(1,obj.num_layers);   % a1,...,aL
    z = cell(1,obj.num_layers-1); % z2,...,zL
    a{1} = data_x; aL = data_x;
    for s = 1:obj.num_layers-1
        w = obj.weights{s}; b = obj.biases{s};
        zs = w*aL+b; aL = sigmoid(zs);
        z{s} = zs; a{s+1} = aL;
    end
end
```

这里，aL 为最终的输出，a 和 z 则存储所有的激活值和带权输入。对应翻译的 Python 代码为

```python
def feedforward(self, data_x):
    a = [];  z = [];
    a.append(data_x); aL = data_x;
    for b, w in zip(self.biases, self.weights):
        zs = np.dot(w, aL)+b
        aL = sigmoid(zs)
        z.append(zs);  a.append(aL);
    return aL, a, z
```

后面不再逐一对应，请参考前面的说明文档。

backward 方法

```
def backprop(self,mini_batch_y,a,z,eta):
    # errors of neurons
    delta = [0]*(self.num_layers-1)
    cost_a = self.cost_derivative(a[-1],mini_batch_y)
    delta[-1] = cost_a*sigmoid_prime(z[-1])
    for i in range(len(z)-1):
        w3 = self.weights[-1-i]
        delta3 = delta[-1-i]
        z2 = z[-1-i-1]
        delta[-1-i-1] = np.dot(w3.T,delta3)*sigmoid_prime(z2)

    # gradient descent: update weights and biases
    m = mini_batch_y.shape[1]
    for level in range(self.num_layers-1):
        delta2 = delta[level]
        a1 = a[level]
        w = self.weights[level]
        b = self.biases[level]
        for i in range(m):
            w = w - eta/m*np.dot(delta2[:,i].reshape(-1,1),a1[:,i].reshape(1,-1))
            b = b - eta/m*delta2[:,i].reshape(-1,1)
        self.weights[level] = w
        self.biases[level] = b
```

注意，这里 [-1] 表示倒数第 1 个，w3.T 表示转置，而 [0]*3 表示 [0, 0, 0]，是一种初始化列表（具指定元素个数）的方法。程序中用到的 cost_derivative 如下

```
def cost_derivative(self,aL,mini_batch_y):
    return (aL-mini_batch_y)
```

SGD 训练

```
def SGD(self, training_data_x, training_data_y, epochs, mini_batch_size, \
            eta, test_data_x=None, test_data_y=None):
    n = training_data_x.shape[1]
    batch_num = int(n/mini_batch_size)

    err = np.zeros(batch_num*epochs,); st = 0;
    for ep in range(epochs):
        kk = random.sample(range(0,n),n)
        for s in range(batch_num):
            # current mini-batch
            id = kk[s*mini_batch_size:(s+1)*mini_batch_size]
            mini_batch_x = training_data_x[:,id]
            mini_batch_y = training_data_y[:,id]

            # feedforward
            aL,a,z = self.feedforward(mini_batch_x)

            # backpropagation
            self.backprop(mini_batch_y,a,z,eta)

```

```
            # compute errors
            err[st] = 0.5*np.mean((aL-mini_batch_y)**2)
            st += 1


        if test_data_x.any():
            # evaluation of test_data
            n_correct = self.evaluate(test_data_x,test_data_y)
            n_test = test_data_x.shape[1]
            print("Epoch {:2d} : {} / {}".format(ep,n_correct,n_test))

    plt.figure(figsize=(6,4))
    plt.plot(err)
```

训练评估函数为

```
def evaluate(self, data_x, data_y):
    data_yp, _, _ = self.feedforward(data_x)
    yp = np.argmax(data_yp, axis=0)
    y = np.argmax(data_y,axis=0)
    return sum(yp == y)
```

`np.argmax` 用来返回数组中最大值的索引，而 axis=0 表示返回行号，因而是每列中的最大值。

主程序

```
#### Libraries
# Standard library
import random
# Third-party libraries
import numpy as np


## Load MNIST
import mnist_loader

training_data_x, training_data_y, \
validation_data_x, validation_data_y, \
test_data_x, test_data_y = mnist_loader.load_data()

## Parameters
sizes = [784,15,10]
epochs = 11
mini_batch_size = 10
eta = 3

## Create a Network object
import network
net = network.Network(sizes)

## Train network with SGD
net.SGD(training_data_x, training_data_y, epochs, mini_batch_size, \
             eta, test_data_x, test_data_y)

```

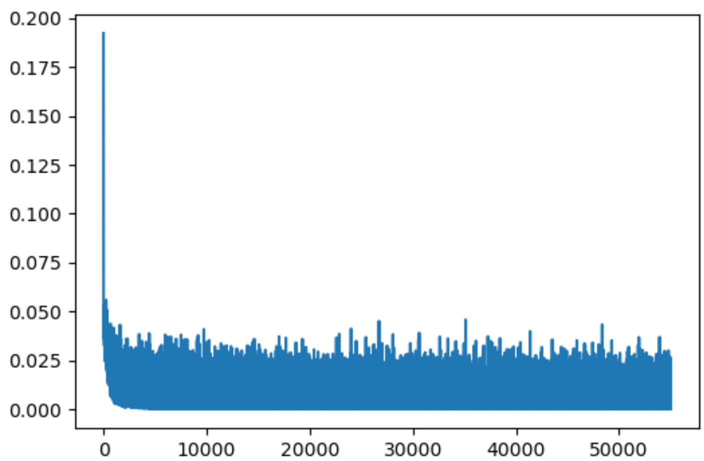
```
## Recognize handwritten digits
num_p, yp, y = net.evaluate(validation_data_x, validation_data_y)
ratio = num_p/len(y)
print("\n Recognize handwritten digits in validation_data \n")
print(" Accuracy = {:.2%} \n".format(ratio))
```

结果如下

```
Epoch  0 : 8887 / 10000
Epoch  1 : 9078 / 10000
Epoch  2 : 9082 / 10000
Epoch  3 : 9152 / 10000
Epoch  4 : 9135 / 10000
Epoch  5 : 9176 / 10000
Epoch  6 : 9089 / 10000
Epoch  7 : 9242 / 10000
Epoch  8 : 9215 / 10000
Epoch  9 : 9247 / 10000
Epoch 10 : 9236 / 10000

 Recognize handwritten digits in validation_data

 Accuracy = 92.94%
```



设 epochs = 11, 可以发现，MATLAB 的速度明显快于 Python：

```
MATLAB: 20.010242 seconds (使用 tic, toc)
Python: 34.8556 s (使用 time.time())
```

Michael Nielsen 的 "Neural Networks and Deep Learning "一书中的程序速度更慢。忽略计算误差、误差图示和 validation_data 准确率的评估，其代码所需时间大约 50 秒。速度慢的根本原因是程序中大量使用了循环（即对样本数据 x 的循环），而没采用矩阵运算。