"""

network.py

~~~~~~~~~~

A module to implement the stochastic gradient descent learning algorithm for a feedforward neural network. Gradients are calculated using backpropagation. Note that I have focused on making the code simple, easily readable, and easily modifiable. It is not optimized, and omits many desirable features.

一种用于实现前馈神经网络的随机梯度下降学习算法的模块。 使用反向传播计算梯度。 请注意，我专注于使代码简单，易于阅读和轻松修改。 它没有被优化，省略了许多理想的特征。

"""

#### Libraries

# Standard library

**import** random

# Third-party libraries

**import** numpy **as** np

**class** **Network(**object**):**

**def** \_\_init\_\_**(**self**,** sizes**):**

"""The list ``sizes`` contains the number of neurons in the respective layers of the network. For example, if the list was [2, 3, 1] then it would be a three-layer network, with the first layer containing 2 neurons, the second layer 3 neurons, and the third layer 1 neuron. The biases and weights for the network are initialized randomly, using a Gaussian distribution with mean 0, and variance 1. Note that the first layer is assumed to be an input layer, and by convention we won't set any biases for those neurons, since biases are only ever used in computing the outputs from later layers."""

列表“大小”包含网络各个层中的神经元数量。 例如，如果列表是[2，3，1]，那么它将是一个三层网络，第一层包含2个神经元，第2层神经元和第3层神经元。 使用平均值为0的方差为1的高斯分布，网络的偏移和权重被初始化初始化。注意，第一层被假设为输入层，按照惯例我们不会为这些神经元设置任何偏差， 因为偏差仅用于计算后期层的输出。

self**.**num\_layers **=** len**(**sizes**)**

self**.**sizes **=** sizes

self**.**biases **=** **[**np**.**random**.**randn**(**y**,** 1**)** **for** y **in** sizes**[**1**:]]**

self**.**weights **=** **[**np**.**random**.**randn**(**y**,** x**)**

**for** x**,** y **in** zip**(**sizes**[:-**1**],** sizes**[**1**:])]**

**def** feedforward**(**self**,** a**):**

"""Return the output of the network if ``a`` is input."""

**for** b**,** w **in** zip**(**self**.**biases**,** self**.**weights**):**

a **=** sigmoid**(**np**.**dot**(**w**,** a**)+**b**)**

**return** a

**def** SGD**(**self**,** training\_data**,** epochs**,** mini\_batch\_size**,** eta**,**

test\_data**=None):**

"""Train the neural network using mini-batch stochastic gradient descent. The ``training\_data`` is a list of tuples ``(x, y)`` representing the training inputs and the desired outputs. The other non-optional parameters are self-explanatory. If ``test\_data`` is provided then the network will be evaluated against the test data after each epoch, and partial progress printed out. This is useful for tracking progress, but slows things down substantially."""

使用迷你批次随机梯度下降训练神经网络。 “training\_data”是表示训练输入和所需输出的元组“（x，y）”的列表。 其他非可选参数是不言自明的。 如果提供了“test\_data”，那么网络将根据每个时期之后的测试数据进行评估，部分进度打印出来。 这对于追踪进度是有用的，但实质上减缓了事情。

**if** test\_data**:** n\_test **=** len**(**test\_data**)**

n **=** len**(**training\_data**)**

**for** j **in** xrange**(**epochs**):**

random**.**shuffle**(**training\_data**)**

mini\_batches **=** **[**

training\_data**[**k**:**k**+**mini\_batch\_size**]**

**for** k **in** xrange**(**0**,** n**,** mini\_batch\_size**)]**

**for** mini\_batch **in** mini\_batches**:**

self**.**update\_mini\_batch**(**mini\_batch**,** eta**)**

**if** test\_data**:**

**print** "Epoch {0}: {1} / {2}"**.**format**(**

j**,** self**.**evaluate**(**test\_data**),** n\_test**)**

**else:**

**print** "Epoch {0} complete"**.**format**(**j**)**

**def** update\_mini\_batch**(**self**,** mini\_batch**,** eta**):**

"""Update the network's weights and biases by applying

gradient descent using backpropagation to a single mini batch.

The ``mini\_batch`` is a list of tuples ``(x, y)``, and ``eta``

is the learning rate."""

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**)**

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 **=** **[**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**):**

"""Return a tuple ``(nabla\_b, nabla\_w)`` representing the

gradient for the cost function C\_x. ``nabla\_b`` and

``nabla\_w`` are layer-by-layer lists of numpy arrays, similar

to ``self.biases`` and ``self.weights``."""

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\_derivative**(**activations**[-**1**],** y**)** **\*** \

sigmoid\_prime**(**zs**[-**1**])**

nabla\_b**[-**1**]** **=** delta

nabla\_w**[-**1**]** **=** np**.**dot**(**delta**,** activations**[-**2**].**transpose**())**

# Note that the variable l in the loop below is used a little

# differently to the notation in Chapter 2 of the book. Here,

# l = 1 means the last layer of neurons, l = 2 is the

# second-last layer, and so on. It's a renumbering of the

# scheme in the book, used here to take advantage of the fact

# that Python can use negative indices in lists.

**for** l **in** xrange**(**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**)**

**def** evaluate**(**self**,** test\_data**):**

"""Return the number of test inputs for which the neural

network outputs the correct result. Note that the neural

network's output is assumed to be the index of whichever

neuron in the final layer has the highest activation."""

test\_results **=** **[(**np**.**argmax**(**self**.**feedforward**(**x**)),** y**)**

**for** **(**x**,** y**)** **in** test\_data**]**

**return** sum**(**int**(**x **==** y**)** **for** **(**x**,** y**)** **in** test\_results**)**

**def** cost\_derivative**(**self**,** output\_activations**,** y**):**

"""Return the vector of partial derivatives \partial C\_x /

\partial a for the output activations."""

**return** **(**output\_activations**-**y**)**

#### Miscellaneous functions

**def** sigmoid**(**z**):**

"""The sigmoid function."""

**return** 1.0**/(**1.0**+**np**.**exp**(-**z**))**

**def** sigmoid\_prime**(**z**):**

"""Derivative of the sigmoid function."""

**return** sigmoid**(**z**)\*(**1**-**sigmoid**(**z**))**