No.1 手写数字辨识

`__init__`方法

- 使用np.random.randn生成均值为0标准差为1的高斯分布，随机给配置神经网络参数，但后面有更好的初始化权重和偏置的方法。
- 每两层之间的w参数本来就是i行j列的矩阵，其中i是l层的神经元个数，j是(l-1)层的神经元个数。而weights表示的是这些所有的矩阵的集合（也就说是个矩阵组）。由于w是两层之间才有，因此weights中包含层数减一个w矩阵。同时通过zip(sizes[:-1], sizes[1:])将l层的神经元个数做行数，(l-1)层的神经元个数做列数。
  而每层的bias本身是一个1维列向量有这层有所多少个神经元，该向量就有多少行。而biases是这些向量的集合。
  Network([1,2,3])
  self.biases共两个向量

```
[array([[ 0.67692681],
       [-0.22281516]]),
 array([[-1.01713186],
       [ 1.62051904],
       [ 0.17539891]])]
```

self.weights共两个向量

```
[array([[-0.5329287],
       [ 0.9173535]]),
 array([[ 0.97626297, -1.19779494],
       [-2.11551506,  0.31956679],
       [-0.45324637,  0.56996521]])]
```

- 注意weights和biases首先是一个列表，这个列表里面的每个元素都是每层的bias或者两层之间的weight的矩阵，这些矩阵是array类型的。在实际使用的时候大多是对列表遍历或解包，从而得到每层和两层之间的weight和bias参数矩阵

feedforward方法

`for b, w in zip(self.biases, self.weights)`
每迭代一次将self.biases中一个1维列向量取出给b，将self.weights的一个i行j列矩阵取出给w。那么b就是第n层的所有偏置，w就是第(n-1)到第n层的所有w参数。
于是该方法构造了两层之间的递推计算公式

SGD方法

`for j in range(epochs):`表示在每个迭代期：
`random.shuffle(training_data)`先打乱训练集，然后再分成小批量数据块。对每一组小批量数据运算以后更新一次参数（使用`update_mini_batch`更新参数）。
`mini_batches`是一个二维列表，列表中的每个元素都是一个列表，最外层的列表表示所有batch

块，每个batch块有时一个包含有固定个数训练元素的列表（即`mini_batch_size`个`training_data`中的元素）。
test_data拖慢进度的原因：每利用一个小批量数据块计算完下一个参数，都会计算当前参数的准确率并打印出来

update_mini_batch

对小批量数据块更新下一组参数。

```
for x, y in mini_batch:  #
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
```

对小批量块中每个训练集元素迭代，并且对每个训练集的元素运用反向传播
`delta_nabla_b`,`delta_nabla_w`这里面分别存储着每个参数b对应的梯度和每个参数w对应的梯度。每调用一次backpro函数会得到某个训练样本多对应的各层w参数和b参数的梯度所对应的矩阵。
根据参数更新公式，某个参数的更新需要对一个小批量快中所有训练样本的该参数的梯度矩阵或向量求和。而每次通过backpro得到的`nabla_b`，`nabla_w`都只是某个训练样本下所有参数的和。因此要对所有训练样本的结果进行累加。
遍历完一次小批量快就可以用公式推得下一组参数。并且这个公式的运用不是一个具体参数的运用而是以每层的参数矩阵为单位的矩阵化运算，对矩阵使用计算公式，对矩阵运用公式的含义也是对每个元素运用公式，但这样实际计算要快一些。

backprop

它的输入是具体的某个训练样本（因为计算的是某个具体训练样本在某组参数下，所有参数的梯度）。和updata方法一样先建立一个空的`nabla_b`，`nabla_w`后面用来存储算出来的该训练样本下的所有参数的梯度矩阵。
第一步完成前向传播

```
for b, w in zip(self.biases, self.weights):
     z = np.dot(w, activation)+b
     zs.append(z)
     activation = sigmoid(z)
     activations.append(activation)
```

计算每一层的输入向量和输出向量分别存储在zs和activations中，
第二步完成后向传播
1.

```
delta = self.cost_derivative(activations[-1], y) * \
     sigmoid_prime(zs[-1])
```

首先计算最后一层的误差（就是误差函数对z的导数）。计算公式就是上面实现的代码，首先利用cost_derivative计算代价函数在最后一层实际输出y处的导数值，这是个向量化运算，对activations[-1]这数组每个元素都执行这个运算。利用`sigmoid_prime`计算处激活函数sigmoid在最后一层的输入zs[-1]处的导数值，这也是一个向量化运算。然后将这两个结果按元素乘以（点

乘）。
计算出误差以后，将误差带入参数梯度的公式，通过误差列向量与倒数第二层的输出a向量的转置矩阵相乘得到最后一层的w参数的梯度矩阵然后将结果存到所有层的w参数梯度矩阵列表的最后一列。同理得到b参数梯度向量。
2.从最后一层往低层逐渐递推得到每一层的w参数梯度矩阵和b参数梯度向量。

```
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())
```

通过负号的下标保证是反向进行的。先按照了l层与l+1之间的误差递推公式先算出误差：第l层的误差向量就等于【第l层到l+1层的w矩阵的转置与l层的误差向量的矩阵相乘结果】再和【激活函数sigmoid在此层z处的导数值】按元素点乘。有了l层的误差值就可以同之前一样的计算此层的w梯度矩阵和b梯度向量了。
直到得到所有层的梯度向量然后返回。可见，返回的`nabla_b`是一个列表，这个列表的每个元素是每两层之间的w参数梯度矩阵，`nabla_w`也是一个列表，这个列表的每个元素是每层的b参数梯度矩阵。`nabla_b`，`nabla_w`是基于某个特定的训练样本算出来的。而一个小批量块中含有m个这种训练样本，是以小批量为单位进行参数更新的，并且根据参数更新公式，某个参数的更新需要对一个小批量快中所有训练样本的该参数的梯度矩阵或向量求和。因此backprop方法求出每个训练样本需要在update方法中求和。

evaluate方法

是对每一组参数测试的时候用的。它的方法就是首先使用feedforward方法，计算出在此组参数，此组输入下的输出。（计算过程就是利用w矩阵乘此层输入向量再加b矩阵最后得一组输出向量）。
feedforward方法最终会算出最后一层的输出y矩阵。然后把每个测试样本的实际输出y和测试集的正确输出组成一个元组。将10000个测试集元素所对应的10000个元组存到一个列表中。判断这10000个元组有多少元组的两个元素是相等的。
使用nargmax函数来将最后实际计算出来的结果转化成0-9.
可以看到这就是对参数应用的一个例子，如果是最终参数，就是可以跑的程序了。

mnist_loader

`training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]`
MNIST中的原始数据格式：一个有两个元素的元组。其中第一个元素是一个ndarray类型的二维数组，该二维数组最外层有50000个元素，每个元素有是一个有784个元素的一维数组，也就是这是一个50000行784列的矩阵代表输入集。第二个元素同理，也是一个ndarray类型的一维数组，第一维50000个元素，每个元素都是一个0-9之间的数字。
总的来说，原始数据中是一个大元组，这个元组有两个元素。第一个元素是一个二维数组，最外层50000个元素里层784个元素。而第二个元素是一个一维数组，数组的元素都是数字类型。并且所有的数组都是ndarray类型的。
神经网络使用的数据格式：是一个列表，列表有50000个元素。每个元素是一个元组。每个元组有2个元素(x, y)。x是一个有784个元素的一维数组，其中存储着784个像素信息。y是一个有10个元素的一维数组，其中这10个元素分别代表数字0-9，哪一位为1就是哪个数字。

总的来说，要求的格式是有50000个元素的列表，每个元素都是一个含有两个元素的元组。元组第一个元素是一个一维数组承载着输入的像素信息，第二个元素也是一个一维数组承载这输出结果。输出结果也有0-9转换为了10位数字来表示。并且所有的数组都是ndarray类型的。
下面一段代码演示了`load_data_wrapper()`对原始数据处理过程，以及最后的数据格式：

```
import numpy as np
import pprint
a=(np.zeros((6, 3), np.int), np.zeros((6,), np.int))
print('原始数据格式：')
pprint.pprint(a)
print('原始数据提取出来的像素信息')
pprint.pprint([x for x in a[0]])
print('实际上进入神经网络的像素信息数组是这种形式:')
pprint.pprint([np.reshape(x, (3,1)) for x in a[0]])
>>>
原始数据格式：
(array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]]),
 array([0, 0, 0, 0, 0, 0]))
原始数据提取出来的像素信息
[array([0, 0, 0]),
 array([0, 0, 0]),
 array([0, 0, 0]),
 array([0, 0, 0]),
 array([0, 0, 0]),
 array([0, 0, 0])]
实际上进入神经网络的像素信息数组是这种形式:
[array([[0],
       [0],
       [0]]),
 array([[0],
       [0],
       [0]]),
 array([[0],
       [0],
       [0]]),
 array([[0],
       [0],
       [0]]),
 array([[0],
       [0],
       [0]]),
 array([[0],
       [0],
       [0]])]
```

整体流程

1.首先靠mnist_loader模块里的函数将MNIST训练集转化成神经网路需要的格式，并将训练集，验证集和测试集分别存储在`training_data`、`validation_data`、`test_data`里面。其中`training_data`用来给神经网络做训练传递给SGD方法的第一个参数。`validation_data`用来

优化超参数，暂时不用。`training_data`用来测试神经网络性能传递给SGD方法的最后一个参数。
2.调用Network方法，说明总层数和每层的神经元个数。
3.调用SGD方法开始训练。
第一步是对一个迭代器epoch做运算。在每一个epoch先打乱训练集顺序。然后对对每一个小批量块进行迭代，每次迭代都对一个小批量块使用updata方法更新一次参数，并把新参数存储到类的实例属性中。
在update方法中是对一个小批量快处理并更新参数。具体处理如下。将一个小批量快包含的所有训练样本（训练集的元素）迭代，对每个训练样本得到该参数下的所有参数的梯度储存在`nabla_b`和`nabla_w`中，具体计算过程是backpro方法利用反向传播完成的。根据参数更新公式，某个参数的更新需要对一个小批量快中所有训练样本的该参数的梯度矩阵或向量求和。因此在update方法中利用`nabla_b`和`nabla_w`的累加，计算每个参数在所有训练样本中所对应的梯度的和。处理完此小批量的所有训练样本以后，更新一次weights和biases实例属性中的参数。
执行完update方法以后，返回SGD方法中，再次下一个小批量块，直到此迭代器epoch的所有小批量块都迭代完毕（本质就是训练集中所有元素都运算了一遍，因为一个epoch就是遍历完所有训练样本）。如果要求测试然后使用evaluate方法对10000个测试集测试当前这组参数的性能，并放回测试结果。然后继续迭代下一个迭代器epoch直到所要求的epoch次数迭代完成结束程序。

参考文献

https://blog.csdn.net/qq_41185868/article/details/79039604
https://blog.csdn.net/qq_17105473/article/details/72823462

星期二, 30. 十月 2018 04:48下午