人工智能导论大作业

10153903015 杜云滔

文件介绍

代码介绍

1. **ReadDigit.py**

   将二进制文件展示成图片。使用`ReadDigit`这个类实现，可使用类函数`showPic ()`指定显示哪一张图片，方便进行可视化查看。

2. **save_data.py**

   将解析出的二进制文件使用`numpy.ndarray`进行存储，使用`pickle`库函数进行序列化，并将image和label保存为一个数组，最后保存在data文件夹中，以`test.pkl`和`train.pkl`进行存储，方便后续操作。

   需要注意的是，这里没有保存原始图像的像元值，而是进行了归一化。

3. **Network.py**

   自己实现的前馈神经网络源代码。其中使用`Network`类初始化神经网络的输入向量和层数，使用`SGD()`梯度下降进行训练。

4. **TF.py**

   使用TensorFlow框架实现了两层的卷积神经网络识别手写数字。

数据介绍

1. MNIST_data

   存放原始的MNIST数据集以及解压后的数据。

2. data

   存放经过预处理后生成矩阵的`pkl`序列化数据。

3. pic.png

手写图像的实例。

jupyter notebook

1. network.ipynb

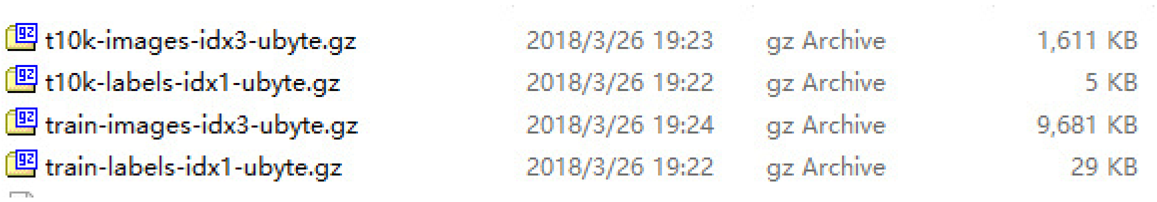
   处理反馈神经网络的代码。

2. result

   存放最后结果（反馈神经网络、卷积神经网络）的HTML文件。

数据介绍

选择[MINIST数据集]进行实验，从官网下载数据后有四个压缩包，如图所示：

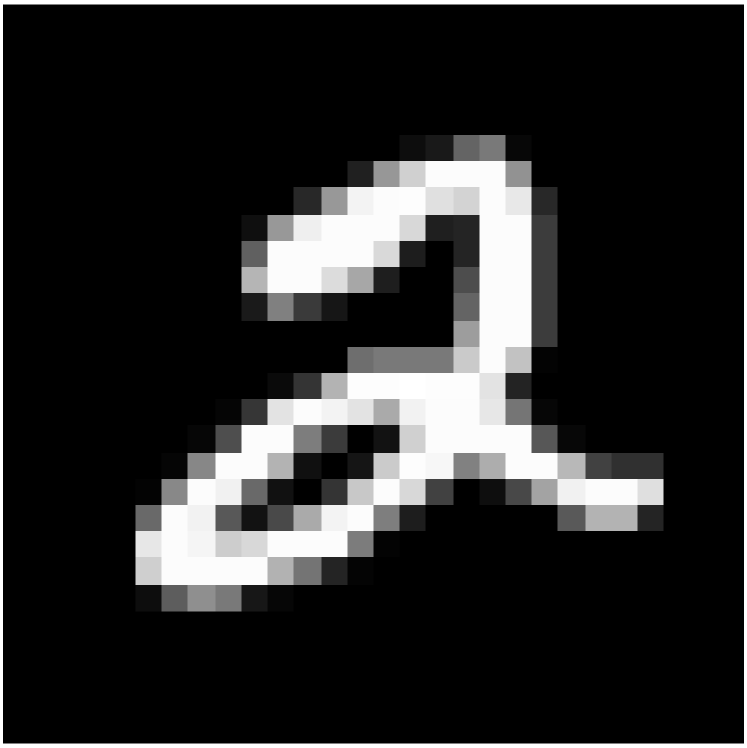


t10k-images-idx3-ubyte.gz	2018/3/26 19:23	gz Archive	1,611 KB
t10k-labels-idx1-ubyte.gz	2018/3/26 19:22	gz Archive	5 KB
train-images-idx3-ubyte.gz	2018/3/26 19:24	gz Archive	9,681 KB
train-labels-idx1-ubyte.gz	2018/3/26 19:22	gz Archive	29 KB

分别为训练集images/label（60000个样本），测试集images/label（10000个样本）。

文件解析

根据官网的描述，首先需要将二进制文件解释成图片或数组，其中包含magic number，offset，type，value，description等信息，网上已经有很多解释，本文参考[这篇文章]对其进行解析，使用`ReadDigit.py`进行可视化展示（具体细节见源码），展示结果如图所示：



序列化

`save_data.py`将其序列化后的`numpy.ndarray`保存为`.pkl`文件。这里需要将解析出来的图像数组和label使用list关联起来，然后再将其做**归一化**处理（实验验证，不进行归一化，神经网络很难训练），最后使用`pickle`库函数将其保存在磁盘中待处理。

数据大小

最后解析出来存储到磁盘中，每一张图片为28*28的`numpy.ndarray`矩阵，在训练集上一共有6000个这样的矩阵，label则为(6000,1)的行向量，表示对应的值。在测试集中，有大小相同的1000个矩阵，label对应的为(1000,1)的行向量。

前馈神经网络实现

本文首先自己实现了简单的前馈神经网络，从公式推导、代码介绍、梯度下降等方面来介绍。

公式推导

符号说明

w_{jk}^l:表示从（$l-1$）层的第k个神经元到第l层的第j个神经元的连接上的权重。虽然从直观上不太好理解为什么要这样表示（通常应该表示为w_{kj}^l），但请先接受这种写法。可以对相邻两层的所有权重用矩阵的形式表示为w^l。

σ：表示激活函数，本文都使用[Sigmoid function]。

b_j^l：表示第l层j神经元的偏置，可以对同一层的神经元表示为b^l，记为偏置向量。

a_j^l：表示第l层j神经元的激活值，可以对同一层的神经元表示为a^l，记为激活向量。 由BP神经网络的定义可得：$a^l = \sigma(w^l a^{l-1} + b^l)$。

z^l：表示带权输入，$z^l = w^l a^{l-1} + b^l \quad a^l = \sigma(z^l)$。

C：表示代价函数，定义为$C = \frac{1}{2n}\sum ||y(x) - a^L(x)||^2$，其中$y(x)$表示每个样本的真实输出，$L$表示神经网络的总层数。

代价函数

BP神经网络的向前传播很简单，就使用之前提到的矩阵形式就可以计算，当我们初始化所有权重和偏置时，得到的结果输出与目标输出肯定有较大差距，我们使用**代价函数**来度量这种差距。定义如下：

$$C = \frac{1}{2n}\sum ||y(x) - a^L(x)||^2$$

那么，当输入和输出固定时，C就是关于w和 b的函数，我们需要对其进行求偏导，以此来更新代价函数。

我们需要对代价函数进行如下定义（假设）：

1. 代价函数可以写成一个在每个训练样本x上的代价函数C_x的均值$C = \frac{1}{n}\sum_x C_x$。
2. 将C仅看成输出激活值a^L的函数。

以下公式，不加说明，C都指特定的C_x。

反向传播的四个方程

反向传播其实就是对权重和偏置变化影响函数过程的理解。最终就是需要计算$\frac{\partial C}{\partial w_{jk}^l}$和$\frac{\partial C}{\partial b_j^l}$。

我们首先定义一个中间量$\delta_j^l = \frac{\partial C}{\partial z_j^l}$，表示为第$l$层第$j$个神经元的误差，然后将$\delta_j^l$关联到$\frac{\partial C}{\partial w_{jk}^l}$和$\frac{\partial C}{\partial b_j^l}$。

这里可能会感到疑惑，为什么会定义这样一个误差呢？ 我们想象，当在一个神经元的带权输入上增加一个很小的变化Δz_j^l，使得神经元输出由$\sigma(z_j^l)$变为$\sigma(z_j^l + \Delta z_j^l)$，那么这个变化将会向网络后面的层进行传播，最终导致整个代价产生$\frac{\partial C}{\partial z_j^l}\Delta z_j^l$的变化。因此，这里有一种启发式的认识，$\frac{\partial C}{\partial z_j^l}$是神经元误差的度量：

$\delta_j^l = \frac{\partial C}{\partial z_j^l}$

在给出方程严谨的证明前，我们不妨从直观上看一下这些方程，这有助于我们的进一步理解。

- **输出层误差方程：**
  - $\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$
  - 右边的第一个项$\frac{\partial C}{\partial a_j^L}$表示代价随着第$j$个输出激活值的变化而变化的速度。第二项刻画了在$z_j^l$处激活函数$\sigma$变化的速度，可以理解为$\Delta a_j^l$。
  - 注意到这个式子的每一部分都是很好计算的。我们如果已知了一个代价函数和激活函数，那么在前向传播中就可以算得每一个δ_j^L。
  - 用矩阵的形式表示第一个式子则更加简单和美妙，注意$\odot$表示矩阵对应元素相乘：
  - $\delta^L = \nabla_a C \odot \sigma'(z^L)$
- **使用下一层的误差来表示当前层的误差：**
  - $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$
  - 当我们知道$l+1$层的误差δ^{l+1}，当我们应用转置的权重矩阵$(w^{l+1})^T$，我们可以凭直觉理解为它是沿着网络反向移动误差，给我们**度量在l层输出误差的计算方法。**
  - 然后，使用hadamard乘积运算，让误差通过l层的激活函数反向传递回来并给出在第l层带权输入的误差δ。
  - 通过组合前两个公式，我们可以计算出任意一层的带权输入误差。
- **代价函数关于网络中任意偏置的改变率：**
  - $\frac{\partial C}{\partial b_j^i} = \delta_j^l$
  - 通过这个方程我们发现，我们需要计算的$\frac{\partial C}{\partial b_j^i}$与$\delta_j^l$完全一致。
- **代价函数关于任何一个权重的改变率：**
  - $\frac{\partial C}{\partial w_{jk}^i} = a_k^{l-1}\delta_j^l$
  - 这告诉我们如何求$\frac{\partial C}{\partial w_{jk}^i}$。其中$a_k^{l-1}$和 δ_j^l我们都已经知道如何计算了，便于理解，我们可以将其化简为：
  - $\frac{\partial C}{\partial w} = a_{in}\delta_{out}$

- 我们发现，当激活值a_{in}很小时，$\frac{\partial C}{\partial w}$也会变得很小。这时候，我们就说权重缓慢学习，表示在进行梯度下降时，这个权重不会改变太多。

通过之前的式子，我们可以发现，如果输入神经元激活值很低，或者输出神经元已经饱和了，权重会学习的非常缓慢。这可以帮助我们选择激活函数。例如，我们可以选择一个不是sigmoid函数的激活函数，使得σ'总是正数，不会趋近于0，这样会防止原始的S型神经元饱和时学习速率下降的情况。

四个基本方程的推导

总结下来一共有四个重要公式：

1. $\boldsymbol{\delta^L = \nabla_a C \odot \sigma'(z^L)}$
   - $\because \delta_j^L = \frac{\partial C}{\partial z_j^L}$
   - $\therefore \delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$
2. $\boldsymbol{\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)}$
   - $\because \delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l}$，表示这一层的神经元对下一层都有影响
   - $\therefore \delta_j^l = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}$
   - $\because z_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}$
   - $\therefore \frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$
   - 带入可得：$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l)$
3. $\boldsymbol{\frac{\partial C}{\partial b_j^i} = \delta_j^l}$
   - $\because b_k^l = z_k^l - \sum_j w_{kj}^l \sigma(z_j^{l-1})$
   - $\therefore \delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial b_j^l} \frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial C}{\partial b_j^l}$
4. $\boldsymbol{\frac{\partial C}{\partial w_{jk}^i} = a_k^{l-1} \delta_j^l}$
   - $\because z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$
   - $\therefore \frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \Rightarrow \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \frac{\partial C}{\partial z_j^l}$
   - $\therefore \frac{\partial C}{\partial w_{jk}^i} = a_k^{l-1} \sigma_j^l$

首先我们可以通过第一个公式算出δ^L，然后利用第二个公式的递推关系可以算出所有的δ，这样，我们就可以很轻松的算出我们想要的每一个$\frac{\partial C}{\partial b_j^i}$以及$\frac{\partial C}{\partial w_{jk}^i}$。

在反向传播中，为了减少计算量，很常见的方法是使用随机梯度下降。思想也很简单，每一个样本都需要进行参与求导实在是计算量太大，但我们可以只去一小部分来进行更新权重，多算几次取平均。

梯度下降

我们使用梯度下降可以加快神经网络的学习，具体原理不再赘述，可以参考我自己写的这篇博客。

小结

我们使用Mini-batch BGD方法来进行BP神经网络训练，具体步骤为：

1. 输入训练样本集合
2. 对每个训练样本x：设置对应的输入激活a_x^1，并进行：
    - 前向传播：对每个$l = 2, 3, 4..., L$,计算z_x^l
    - 输出误差$\sigma_x^l = \nabla_a Cx \odot \sigma'(z_x^L)$
    - 反向转播误差：对每个$l = L-1, L-2, \ldots, 2$，计算$\delta_x^l = ((w^{l+1})^T \delta_x^{l+1}) \odot \sigma'(z_x^l)$
3. 梯度下降：根据$w^l \rightarrow w^l - \frac{\eta}{m}\sum_x \delta_x^l (a_x^{l-1})^T$和$b^l \rightarrow b^l - \frac{\eta}{m}\sum_x \delta_x^l$更新权值和偏置。

代码介绍

1. load_data_wrapper（）

    将序列化的数据加载到程序中，并将28*28的数组`reshape`为(784,1)的向量，方便作为输入向量。

2. Network.__init__()

    将输入的神经网络的参数（每一层多少个神经元）转化为对应的w和b，使用**标准正太分布**进行随机初始化。

3. Network.feedforward()

    前向传播使用w和b计算每一层的激活值。

4. Network.SGD()

    使用Mini-batch BGD随机选取小样本进行训练。

5. Network.backprop()

反向传播计算每一层的Δw和Δb，根据学习速率从后往前计算。

6. Netwo.evaluate()

   对每一层计算出的结果进行评估准确率。

架构信息

- 选择了自己实现的前馈神经网络进行训练。
- 包含2层神经网络，第一层有100个神经元，第二层有100个神经元，输出层有10个神经元。
- 参数w和 b使用标准正太分布进行随机初始化。
- 损失函数选择的均方误差$C = \frac{1}{2n}\sum||y(x) - a^L(x)||^2$。
- 激活函数选择Sigmoid function。
- 评价指标为最终的准确率，能达到97%左右。
- 迭代次数（epoch）为50次，每五次输出一次，结果如下（只输出了最后一次的w和 b）

实现结果

```
net.SGD(training_data, 50, 10, 3.0, test_data=test_data)
```

```
Epoch 0 : 8123 / 10000
Epoch 5 : 8671 / 10000
Epoch 10 : 9616 / 10000
Epoch 15 : 9669 / 10000
Epoch 20 : 9670 / 10000
Epoch 25 : 9666 / 10000
Epoch 30 : 9691 / 10000
Epoch 35 : 9692 / 10000
Epoch 40 : 9714 / 10000
Epoch 45 : 9716 / 10000
the weight is [array([[ 0.91375296, -0.18528853, -1.64291544, ...,  0.00667428,
        -1.43317705, -0.29134309],
       [ 0.95753864,  0.98811756,  1.92625232, ..., -0.05913896,
         1.65162323,  0.42728173],
       [-0.56106008,  0.42502855,  0.34442562, ...,  0.31648228,
         0.33775879,  0.4689368 ],
       ...,
       [-0.6459578 , -1.03017731, -0.45813785, ..., -2.05104017,
        -0.57547295, -0.09048588],
       [ 2.0241108 , -0.37423648, -0.05921099, ..., -0.92418565,
```

卷积神经网络实现

架构信息

1. 选择了TensorFlow框架进行试验。
2. 构建了两层的卷积神经网络，第一层32个神经元，第二层64个神经元，全连接层有1024个神经元，最后是输出层。
3. 初始值w和b用一个较小的正数来初始化偏置项，以避免神经元节点输出恒为0的问题，这里使用`tf.truncated_normal`进行初始化，使得值都小于2。
4. 使用交叉熵作为代价函数。
5. 迭代次数（epoch）为20000次，每1000次输出一次，最终准确率达到了99%。
6. 最后使用accuracy作为评价指标，结果如下图所示。

实现结果

```
step 0, training accuracy 0.22
step 1000, training accuracy 0.94
step 2000, training accuracy 1
step 3000, training accuracy 1
step 4000, training accuracy 1
step 5000, training accuracy 1
step 6000, training accuracy 0.98
step 7000, training accuracy 1
step 8000, training accuracy 0.98
step 9000, training accuracy 1
step 10000, training accuracy 1
step 11000, training accuracy 1
step 12000, training accuracy 1
step 13000, training accuracy 0.98
step 14000, training accuracy 1
step 15000, training accuracy 1
step 16000, training accuracy 1
step 17000, training accuracy 1
step 18000, training accuracy 1
step 19000, training accuracy 1
test accuracy 0.9917
```

参考资料

1. 读取mnist数据集并保存成图片
2. Neural Networks and Deep Learning

3. MNIST For ML Beginners