第二章作业

一、多分类问题的四个反向传播方程（损失函数为交叉熵损失函数，使用 ReLU 激活函数）

一.多分类问题的四个反向传播方程

① 输出层使用softmax函数. : $a_i^L = \frac{e^{z_i^L}}{\sum_{j=1}^{K} e^{z_j^L}}$

a_i^L表示输出层经过softmax激活后的值

z_i^L表示softmax输入的值.

K表示输出层神经元个数.

损失函数为交叉熵损失函数:

$Loss = E = -\sum_{j=1}^{K} y_j \ln a_j^L$

y为label 是one-hot向量.

y_j表示y向量第j个值.

∵ y为one-hot向量 ∴ 仅有一个值为1, 其余为0

假设对于y有 $y_i=1$. 则 $y_j=0$. j为除i的其余所有值

$E = -\ln a_i^L = -z_i^L + \ln \sum_{j=1}^{K} e^{z_j^L}$

设δ^L为中间变量保存第L层误差. δ^L则为输出层误差. δ_j^L表示输出层误差第j个

$\delta_i^L = \frac{\partial E}{\partial z_i^L} = -1 + \frac{e^{z_i^L}}{\sum_{j=1}^{K} e^{z_j^L}} = -1 + a_i^L$

同理

$\delta_j^L = \frac{\partial E}{\partial z_j^L} = 0 + a_j^L$

$\Rightarrow \delta_j^L = a_j^L - y_j$, j为1...K任意值

⇓ 向量表达形式

$\delta^L = a^L - y$

∴ $\boxed{\text{(BP1) 输出层误差的向量表达形式为 } \delta^L = a^L - y}$

② (BP2) 使用下一层的误差 δ^{l+1} 表示当前层误差 δ^l

除输出层外,中间隐层的激活函数使用ReLU激活函数 即 $\sigma(z) = \begin{cases} 0, & z \le 0 \\ z, & z > 0 \end{cases}$

$z_j^{l+1} = \sum_{k=1}^{K} W_{jk}^l a_k^l + b_j^l$

z_j^{l+1}表示第l+1层第j个神经元z值.

W_{jk}^l表示第l层第k个神经元发送到第l+1层j个神经元的权值

b_j^l表示第l层发送到第l+1层j个神经元的偏置.

$\delta_j^l = \frac{\partial E}{\partial z_j^l} = \sum_{k=1} \frac{\partial E}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_{k=1} \delta_k^{l+1} \cdot \frac{\partial z_k^{l+1}}{\partial z_j^l}$

$z_k^{l+1} = \sum_{t=1} W_{kt}^l a_t^l + b_k^l = \sum_{t=1} W_{kt}^l \sigma(z_t^l) + b_k^l$

这里t表示l层的神经元

这里k表示l+1层的第k个神经元.

$\frac{\partial z_k^{l+1}}{\partial z_j^l} = W_{kj}^l \cdot \sigma'(z_j^l)$

$\Rightarrow \delta_j^l = \sum_{k=1} \delta_k^{l+1} \cdot W_{kj}^l \sigma'(z_j^l) \xrightarrow{\text{向量表达形式}} \delta^l = (W^l \cdot \delta^{l+1}) \odot \sigma'(z^l)$ (BP2)

③ (BP3) 代价函数关于偏置bias的偏导. $\frac{\partial E}{\partial b^{l-1}}$

$$z_j^l = \sum_{k=1} W_{jk}^{l-1} a_k^{l-1} + b_j^{l-1}$$

$$\frac{\partial E}{\partial b_j^{l-1}} = \frac{\partial E}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^{l-1}} = \delta_j^l * 1 = \delta_j^l$$

$$\Rightarrow \frac{\partial E}{\partial b^{l-1}} = \delta^l$$

④ (BP4) 代价函数关于权重的偏导 $\frac{\partial E}{\partial W^{l-1}}$

$$z_j^l = \sum_{k=1} W_{jk}^{l-1} a_k^{l-1} + b_j^{l-1}$$

$$\frac{\partial E}{\partial W_{jk}^{l-1}} = \frac{\partial E}{\partial z_j^l} \frac{\partial z_j^l}{\partial W_{jk}^{l-1}}$$

$$= \delta_j^l \cdot a_k^{l-1}$$

$$\frac{\partial E}{\partial W_j^{l-1}} = a^{l-1} \cdot \delta_j^l \Rightarrow \frac{\partial E}{\partial W^{l-1}} = a^{l-1} (\delta^l)^T \quad \text{(BP4)}$$

二、手写数字识别

1、代码解析

（1）mnist_reader.py 读取 mnist.pkl.gz 中数据并将其转换为常用格式

```python
import pickle
import gzip
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt


# 解压数据集并读取
def load_data():
    file = gzip.open('mnist.pkl.gz', 'rb')
    train_data, validation_data, test_data = pickle.load(file, encoding="bytes")
    # train_data (50000*784,5000*1)
    # print(train_data[0].shape)
    file.close()
    return train_data, validation_data, test_data


# 处理读取的初始数据，转换为常用格式
def raw_data_process():
    raw_tr_data, raw_val_data, raw_te_data = load_data()
    # 训练集
    train_inputs = [np.reshape(x, (784, 1)) for x in raw_tr_data[0]]  # input 784*50000
    train_labels = [one_hot_transform(x) for x in raw_tr_data[1]]  # label 10*50000
    # input 和 label 一一对应
    train_data = list(zip(train_inputs, train_labels))
    # print("训练集大小：{}".format(len(train_labels)))
    # 验证集
    validation_inputs = [np.reshape(x, (784, 1)) for x in raw_val_data[0]]  # 784*10000
    validation_labels = [one_hot_transform(x) for x in raw_tr_data[1]]  # 输出 10*10000
    validation_data = list(zip(validation_inputs, validation_labels))
    # 测试集
    test_inputs = [np.reshape(x, (784, 1)) for x in raw_te_data[0]]  # 784*10000
    test_labels = [one_hot_transform(x) for x in raw_te_data[1]]  # 输出 10*10000
    test_data = list(zip(test_inputs, test_labels))
```

```
    return train_data, validation_data, test_data


# 生成 one-hot 向量
def one_hot_transform(j):
    # shape 10*1
    label_y = np.zeros((10, 1))
    label_y[j] = 1.0
    return label_y


# 使用 PIL 图像 api，将图像显示出来 784*1 28*28
def show_image(vector):
    vector.resize((28, 28))
    img = Image.fromarray(np.uint8(vector * 255)).convert("1")  # 二值化
    plt.imshow(img)
    plt.show()


if __name__ == '__main__':
    train_data, validation_data, test_data = raw_data_process()
    # train_data[0] 784*50000 train_data[1] 10*50000
    # print(train_data[0][1])  # 784*1
    show_image(train_data[0][1])
```

（2）network.py 构建输入层-中间隐层-输出层网络结构，实现前向传播、反向传播和随机梯度下降代码

```
import numpy as np
import random


# ReLU 激活函数
def relu(z):
    z[z < 0] = 0
    return z


# ReLU 激活函数的导数
def d_relu(z):
    z[z > 0] = 1
    z[z <= 0] = 0
    return z


```

```
# softmax 函数
def softmax(z):
    t = np.exp(z)
    a = np.nan_to_num(np.exp(z) / np.sum(t))
    return a


# 神经网络的类
class Network(object):
    # 构造函数初始化网络
    def __init__(self, sizes):   # sizes(784, 15, 10) 输入 784 隐层 15 输出 10
        # 神经网络层数
        self.layer_nums = len(sizes)
        self.sizes = sizes
        # randn(j, i) 可以生成 y 行 x 列的随机数矩阵，是均值为 0,标准差为 1 的高斯分布
        # bias 向量维度由下一层神经元个数决定 15*1 10*1
        self.biases = [np.random.randn(i, 1) for i in sizes[1:]]
        # weights 矩阵维度由输入层和下一层共同决定，15*784 10*15 w^T 方便运算
        self.weights = [np.random.randn(j, i) for i, j in zip(sizes[:-1], sizes[1:])]  # [784 15] [15,10]

    # 前向传播
    # z1=W1^T*x+B1 a1 = relu(z1) z2=W2^T*a1+B2 a2= softmax(z2) a2=y^
    def forward(self, a):
        # 中间隐藏层的激活函数选择 relu，输出层的激活函数为 softmax
        num_forward = 1
        for w, b in zip(self.weights, self.biases):
            z = np.dot(w, a)+b
            if num_forward < (self.layer_nums - 1):  # relu 激活
                a = relu(z)
                num_forward = num_forward + 1
            else:
                a = softmax(z)
        return a

    # 随机梯度下降(训练数据,迭代次数,batch 大小,学习率,是否有测试集)
    def SGD(self, train_data, epochs, mini_batch_size, learning_rate, test_data=None):
        # 迭代过程
        print("Training.........")
        n = len(train_data)  # 训练数据大小
        for j in range(epochs):
            # 打乱训练集
```

```python
            random.shuffle(train_data)
            # mini_batches 是分批后的 mini_batch 列表
            mini_batches = [train_data[k:k+mini_batch_size] for k in range(0, n, mini_batch_size)]
            # 每个 mini_batch 都更新一次,重复整个数据集
            self.update_mini_batch(mini_batches, learning_rate)
            # 若有测试数据,则在屏幕上打印训练进度
            if test_data:
                len_test = len(test_data)
                correct_num = self.evaluate(test_data)
                print("Epoch{0}:{1}/{2}".format(j, correct_num, len_test))
            else:
                print("Epoch {0} complete:".format(j))

    # 更新 mini_batch
    def update_mini_batch(self, mini_batches, learning_rate):
        for mini_batch in mini_batches:
            # 存储对于各个参数的偏导，格式和 self.biases 和 self.weights 是一样的
            nabla_b = [np.zeros(b.shape) for b in self.biases]
            nabla_w = [np.zeros(w.shape) for w in self.weights]
            eta = learning_rate / len(mini_batch)
            # mini_batch 中的一个实例调用梯度下降得到各个参数的偏导 mini_batch(x,y)元组
            for x, y in mini_batch:
                # 从一个实例得到的梯度
                delta_nabla_b, delta_nabla_w = self.backprop(x, y)  # 依次取出 mini_batch 中的（x,y)输入 backprop
                # nabla_w,nabla_b 表示整个 mini_batch 所有训练样本的总代价函数梯度
                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)]
            # 每一个 mini_batch 更新一下参数
            self.biases = [b - eta * nb for b, nb in zip(self.biases, nabla_b)]
            self.weights = [w - eta * nw for w, nw in zip(self.weights, nabla_w)]

    # 反向传播(对于每一个实例)
    def backprop(self, x, y):
        # 生成权重矩阵形状和偏置矩阵形状的零矩阵用于存放每层的梯度
        nabla_b = [np.zeros(b.shape) for b in self.biases]
```

```python
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # 前向传播
        activation = x  # activation 存储激活值
        activations = [x]  # 存储每层的激活值 a
        z_save = []   # 存储前向传播的 z
        current_layer = 1  # 当前层数
        for w, b in zip(self.weights, self.biases):
            z = np.dot(w, activation)+b
            z_save.append(z)
            # 最后一层使用 softmax,前几层使用 relu
            if current_layer < (self.layer_nums - 1):
                activation = relu(z)
                current_layer = current_layer + 1
            else:
                activation = softmax(z)
            activations.append(activation)
        # 计算 loss 反向传播
        delta = self.d_cost(activations[-1], y)  # 输出层的 a^L
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # 倒数第二层开始求偏导
        for l in range(2, self.layer_nums):
            delta = np.dot(self.weights[-l+1].transpose(), delta)*d_relu(z_save[-l])
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return nabla_b, nabla_w

    # 代价函数偏导
    def d_cost(self, output_activations, y):  # 输出的激活值 a, label_y, z^L
        return output_activations-y   # 交叉熵代价函数  E = -ln(a^L) ai^L=softmax(zi^L) E 对 ai^L 求偏导

    # 验证准确率
    def evaluate(self, test_data):
        # 神经网络的输出结果是输出层激活值最大的一个神经元所对应的结果，使用 numpy 的 argmax 方法来找到该输出层神经元的编号
        # 将测试得到的结果以二元组(神经网络判断结果,正确结果) 形式存储
        test_result = [(np.argmax(self.forward(x)), np.argmax(y)) for (x, y) in test_data]
        return sum(int(i == j) for (i, j) in test_result)
```

（3）main.py 调用前两个 py 文件函数

```python
# 实现一个手写数字识别程序
import mnist_reader
import network


# 主函数
if __name__ == '__main__':
    # 读取原始数据处理输出 训练集 验证集 测试集
    train_data, validation_data, test_data = mnist_reader.raw_data_process()
    # 初始化神经网络 (784, 15, 10)
    net = network.Network((784, 15, 10))
    # 训练神经网络
    epochs = 10  # 训练次数
    mini_batch_size = 10  # batch 大小
    learning_rate = 0.1  # 学习率
    net.SGD(train_data, epochs, mini_batch_size, learning_rate, test_data=test_data)
    # 测试神经网络
    print("Test times {0}: {1}/{2}(正确识别个数/训练总数)".format(0, net.evaluate(test_data), 10000))
```

2、实验结果

（1）batch_size = 10 learning_rate = 0.1

```
D:\Python\anaconda\anaconda3\envs\pytorch_gpu\python.exe D:/Python/pycharm/pythonProject/numberRecognition/main.py
Training.........
Epoch0:7858/10000
Epoch1:8687/10000
Epoch2:8918/10000
Epoch3:8843/10000
Epoch4:9067/10000
Epoch5:9004/10000
Epoch6:9101/10000
Epoch7:9161/10000
Epoch8:9095/10000
Epoch9:9190/10000
Test times 0: 9190/10000(正确识别个数/训练总数)

Process finished with exit code 0
```

（2）batch_size = 10 learning_rate = 0.5 学习率过大，局部最优解

```
D:\Python\anaconda\anaconda3\envs\pytorch_gpu\python.exe D:/Python/pycharm/pythonProject/numberRecognition/main.py
Training.........
Epoch0:5888/10000
Epoch1:6542/10000
Epoch2:7976/10000
Epoch3:8283/10000
Epoch4:8632/10000
Epoch5:8632/10000
Epoch6:8652/10000
Epoch7:8443/10000
Epoch8:8671/10000
Epoch9:8669/10000
Test times 0: 8669/10000(正确识别个数/训练总数)

Process finished with exit code 0
```