两层的神经网络对 Mnist 数据集进行分类

18051810 陈锐

第一部分：讲解

1. 运行环境:Python 3.7+Pycharm
2. 代码讲解:

1.前向传播函数

```
def affine_forward(x, w, b):
    out = None                          # 初始化返回值为 None
    N = x.shape[0]                      # 重置输入参数 X 的形状
    x_row = x.reshape(N, -1)            # (N,D)
    out = np.dot(x_row, w) + b          # (N,M)
    cache = (x, w, b)                   # 缓存值，反向传播时使用
    return out,cache
```

这一段程序是定义了了一个名为 affine_forward 的函数，其功能就是计算这个公式

$$H=X*W1+b1$$

2.反向传播函数

```
def affine_backward(dout, cache):
    x, w, b = cache                              # 读取缓存
    dx, dw, db = None, None, None                # 返回值初始化
```

```
    dx = np.dot(dout, w.T)                    # (N,D)
    dx = np.reshape(dx, x.shape)              # (N,d1,...,d_k)
    x_row = x.reshape(x.shape[0], -1)         # (N,D)
    dw = np.dot(x_row.T, dout)                # (D,M)
   db = np.sum(dout, axis=0, keepdims=True)   # (1,M)
    return dx, dw, db
```

仿射变换反向传播的最重要的 3 个目的，分别是：①更新参数 w 的值②计算流向下一个节点的数值③更新参数 b 的值。“更新”的时候需要“旧”值，也就是缓存值计算反向传播梯度，通过计算权重和偏差的导数。并把结果存储在梯度字典中。

$$p_k = \frac{e^{f_k}}{\sum_j e^{f_j}} \qquad L_i = -\log(p_{y_i})$$

$$\frac{\partial L_i}{\partial f_k} = p_k - 1(y_i = k)$$

3. 参数初始化，读取 mnist 数据集

```
np.random.seed(1)        # 有这行语句,你们生成的随机数就和我一样了
train_data, train_label, test_data, test_label = H.load_data(r"C:\Users\1\Desktop\MNIST_DATA1", 0.5)
# X = np.mat(X)  # 这三行实现的是将序列转换成矩阵，mat 是numpy 里转换成矩阵的函数
X= np.mat(train_data)
t = np.mat(train_label)
```

```
# 一些初始化参数
t=np.array(list(chain.from_iterable(t.tolist())))
print(t)

input_dim = X.shape[1]      # 输入参数的维度，此处为 1，
num_classes = t.shape[0]    # 输出参数的维度，此处为很多
hidden_dim = 50             # 隐藏层维度，为可调参数
reg = 0.001                 # 正则化强度，为可调参数
epsilon = 0.001             # 梯度下降的学习率，为可调参数
# 初始化 W1，W2，b1，b2
W1 = np.random.randn(input_dim, hidden_dim)     # (2,50)//初始化维度
W2 = np.random.randn(hidden_dim, num_classes)   # (50,4)
b1 = np.zeros((1, hidden_dim))                  # (1,50)
b2 = np.zeros((1, num_classes))
loss_1=[]
```

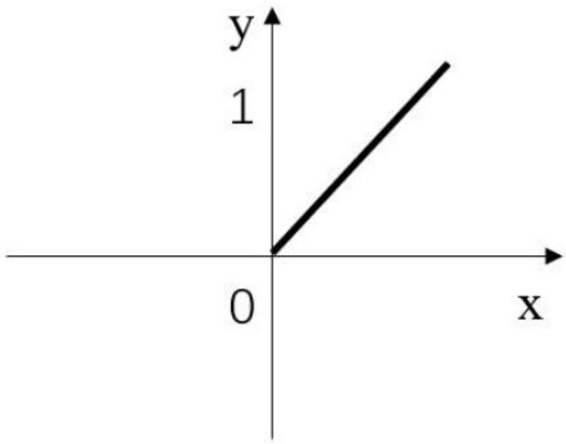
4. **进行 10000 次训练,并且计算出每次训练的 Loss 值**

```
for j in range(10000):   #这里设置了训练的循环次数为 10000
 # ①前向传播
    H,fc_cache = affine_forward(X,W1,b1)                 #第一层前向传播
    H = np.maximum(0, H)                                 #
```

```
激活
    relu_cache = H                                                  #
缓存第一层激活后的结果
    Y,cachey = affine_forward(H,W2,b2)                              #
第二层前向传播
 # ②Softmax 层计算
    probs = np.exp(Y - np.max(Y, axis=1))
    probs /= np.sum(probs, axis=1)          # Softmax 算法实
现
 # ③计算 loss 值
    N = Y.shape[0]                                                  #
值为数据的个数
    print(probs[np.arange(N), t])
# 打印各个数据的正确解标签对应的神经网络的输出
    loss = -np.sum(np.log(probs[np.arange(N), t])) / N  #
计算 loss
    loss_1.append(loss)
    print("第%d 次 loss 值为%f" % (j, loss))
# 打印 loss
 # ④反向传播
    dx = probs.copy()                                               #
以 Softmax 输出结果作为反向输出的起点
```

```
    dx[np.arange(N), t] -= 1                                          #
    dx /= N                                                           #
到这里是反向传播到 softmax 前
    dh1, dW2, db2 = affine_backward(dx, cachey)                       #
反向传播至第二层前
    dh1[relu_cache <= 0] = 0                                          #
反向传播至激活层前
    dX, dW1, db1 = affine_backward(dh1, fc_cache)                     #
反向传播至第一层前
# ⑤参数更新
    dW2 += reg * W2
    dW1 += reg * W1
    W2 += -epsilon * dW2
    b2 += -epsilon * db2
    W1 += -epsilon * dW1
    b1 += -epsilon * db1
 # ④反向传播
    dx = probs.copy()  # 以 Softmax 输出结果作为反向输出的
起点
    dx[np.arange(N), t] -= 1  #
    dx /= N  # 到这里是反向传播到 softmax 前
    dh1, dW2, db2 = affine_backward(dx, cachey)  # 反向传
```

```
播至第二层前
    dh1[relu_cache <= 0] = 0  # 反向传播至激活层前
    dX, dW1, db1 = affine_backward(dh1, fc_cache)  # 反向
传播至第一层前
 # ⑤参数更新
    dW2 += reg * W2
    dW1 += reg * W1
    W2 += -epsilon * dW2
    b2 += -epsilon * db2
    W1 += -epsilon * dW1
    b1 += -epsilon * db1
test=np.mat(test_data)
H, fc_cache = affine_forward(test, W1, b1)  # 仿射
H = np.maximum(0, H)  # 激活
relu_cache = H
Y, cachey = affine_forward(H, W2, b2)  # 仿射
# Softmax
```



ReLU

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

交叉熵损失的求法是求对数的负数。

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\frac{1}{2}\lambda\sum_k\sum_l W_{k,l}^2}_{\text{regularization loss}}$$

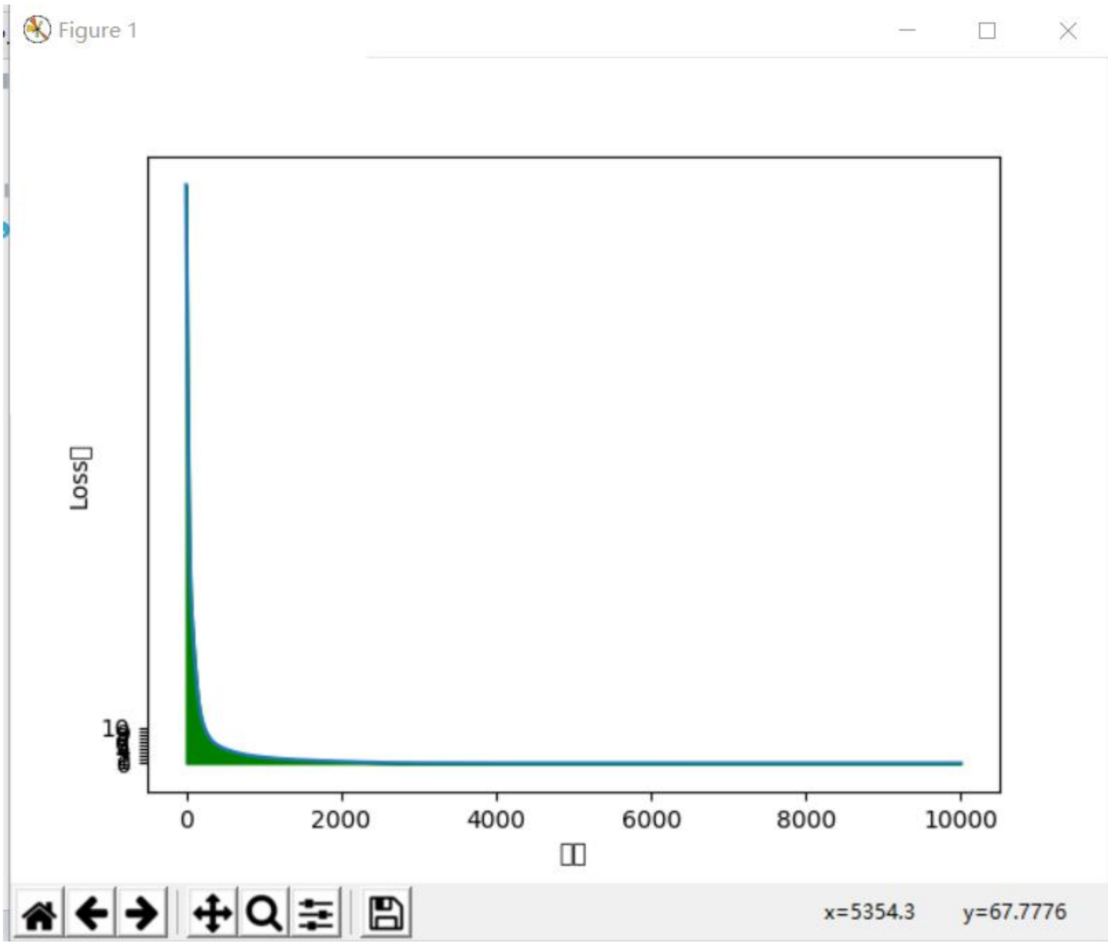
5.　得出正确率以及 Loss 曲线图

```
for k in range(200):
  if(np.argmax(probs[k, :])==t[k]):
        sum=sum+1
print(sum/200)
t=list(range(10000))
pyplot.plot(t,loss_1)
pyplot.xlabel('次数')
pyplot.ylabel('Loss 值')
```

```
pyplot.yticks([0,1,2,3,4,5,6,7,8,9,10])
pyplot.fill_between(t,loss_1,color = 'green')
pyplot.show()
```

第二部分：得到的结果：

Loss 曲线：



最终的 Loss 值：0.001437

```
 1.          1.          1.          0.99338884 1.
 0.99999998 0.99326141 0.99765675 1.          1.
 1.          1.          0.99490439 0.99982918 1.
 0.99994091 1.          1.        ]]
第9999次loss值为0.001437
```

最终正确率:54.35%