Homework 4

Jingwei Zhang 201528013229095

2015-12-1

1 Problem 1

题目1.

(a) 记输出层转移函数为 $f_j(net_1, net_2, \ldots, net_m)$，隐含层转移函数为 $g(net_h)$，用字母 i, h, j 表示输入，中间，输出层节点编~~易得g~~号，用 x_i, y_h, z_j 表示这三层节点的输出.

易得 $g'(net_h)=y_h(1-y_h)$，当 $j'=j$ 时，$\frac{\partial z_j'}{\partial net_j}=z_j(1-z_j)$，当 $j'\neq j$ 时 $\frac{\partial z_j'}{\partial net_j}=-z_{j'}\cdot z_j$

$\therefore$ 令 $f_{j'}=\begin{cases} z_j\cdot(1-z_j), & j'=j \\ -z_{j'}\cdot z_j, & j'\neq j \end{cases}$

则
$$\frac{\partial J(w)}{\partial w_{hj}}=\frac{1}{2}\sum_{j'=1}^{c}\frac{\partial}{\partial w_{hj}}(t_{j'}-z_{j'})^2$$
$$=-\sum_{j'=1}^{c}\left[(t_{j'}-z_{j'})\cdot\frac{\partial f_{j'}}{\partial net_{j'}}\cdot\frac{\partial net_{j'}}{\partial w_{hj}}\right]$$
$$=-\sum_{j'=1}^{c}\left[(t_{j'}-z_{j'})\cdot f_{j'}'\cdot\frac{\partial}{\partial w_{hj}}\sum_{h=1}^{m}(w_{hj'}\cdot y_h)\right]$$
$$=-(t_j-z_j)\cdot f_j'\cdot y_h.$$
$$=-(t_j-z_j)z_j(1-z_j)\cdot y_h.$$

令 $\delta_j=(t_j-z_j)\cdot z_j(1-z_j)$

则 $\Delta w_{hj}=-\eta\frac{\partial J(w)}{\partial w_{hj}}=\eta\delta_j\cdot y_h.$

得隐含层到输出层更新法为 $w_{hj\,new}=w_{hj\,odd}+\Delta w_{hj}$

又
$$\frac{\partial J(w)}{\partial w_{ih}}=\frac{1}{2}\sum_{j'=1}^{m}\frac{\partial}{\partial net_{j'}}(t_{j'}-z_{j'})^2\cdot\frac{\partial}{\partial w_{ih}}\sum w_{h'j'}\cdot y_{h'}$$
$$=-\sum_{j'=1}^{c}(t_j-z_j)\,z_{j'}(1-z_{j'})\cdot\sum_{h'=1}^{m}w_{h'j'}\cdot\frac{\partial g(net_{h'})}{\partial net_{h'}}\cdot\frac{\partial(net_{h'})}{\partial w_{ih}}$$
$$=-\sum_{j=1}^{c}\delta_j\cdot\sum_{h'=1}^{m}w_{h'j}\,g'(net_{h'})\cdot\frac{\partial}{\partial w_{ih}}\sum_{i=1}^{d}w_{i'h'}\cdot x_{i'}$$
$$=-\sum_{j=1}^{c}\delta_j\cdot w_{hj}\cdot y_h(1-y_h)\cdot x_i$$

令 $\delta_h=y_h\cdot(1-y_h)\cdot\sum_{j=1}^{c}w_{hj}\cdot\delta_j$

则 $\Delta w_{ih}=-\eta\frac{\partial J(w)}{\partial w_{ih}}=\eta\delta_h\cdot x_i$

即有 $w_{ih\_new}=w_{ih\_odd}+\Delta w_{ih}.$

(b) 反向传播算法本质上就是对一个~~方差判断~~最小平方误差函数进行优化的过程

通过逐层地对所要更新的权值不断使用链式法则求导，从后向前依次更新每一层的权值，同时，每一层结点的权值，只与前一层的误差有关，这就得到了一个迭代的动态规划算法，~~却~~从最后一层依次向前，使得复杂度大大降低

2 Problem 2

题目2:

步骤:

①初始化权重

②选择输入向量X

③计算X与映射层向量的距离 d_j

④选择最小的 d_j，计 $j^* = \arg\min_j d_j$

⑤选择 j^* 的邻域 $N_j(j^*)$

⑥对 $N_j(j^*)$ 中所有权重，更新如下：

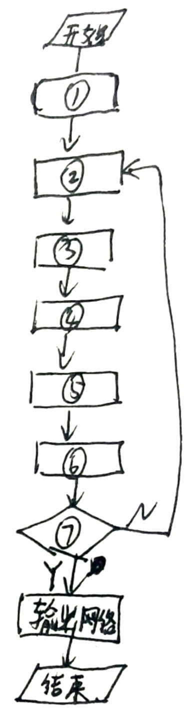
$$\Delta W_{ij} = \eta\, h(j, j^*)\,(x_i - W_{ij})$$

$$W_{ij}(t+1) = W_{ij}(t) + \Delta W_{ij}$$

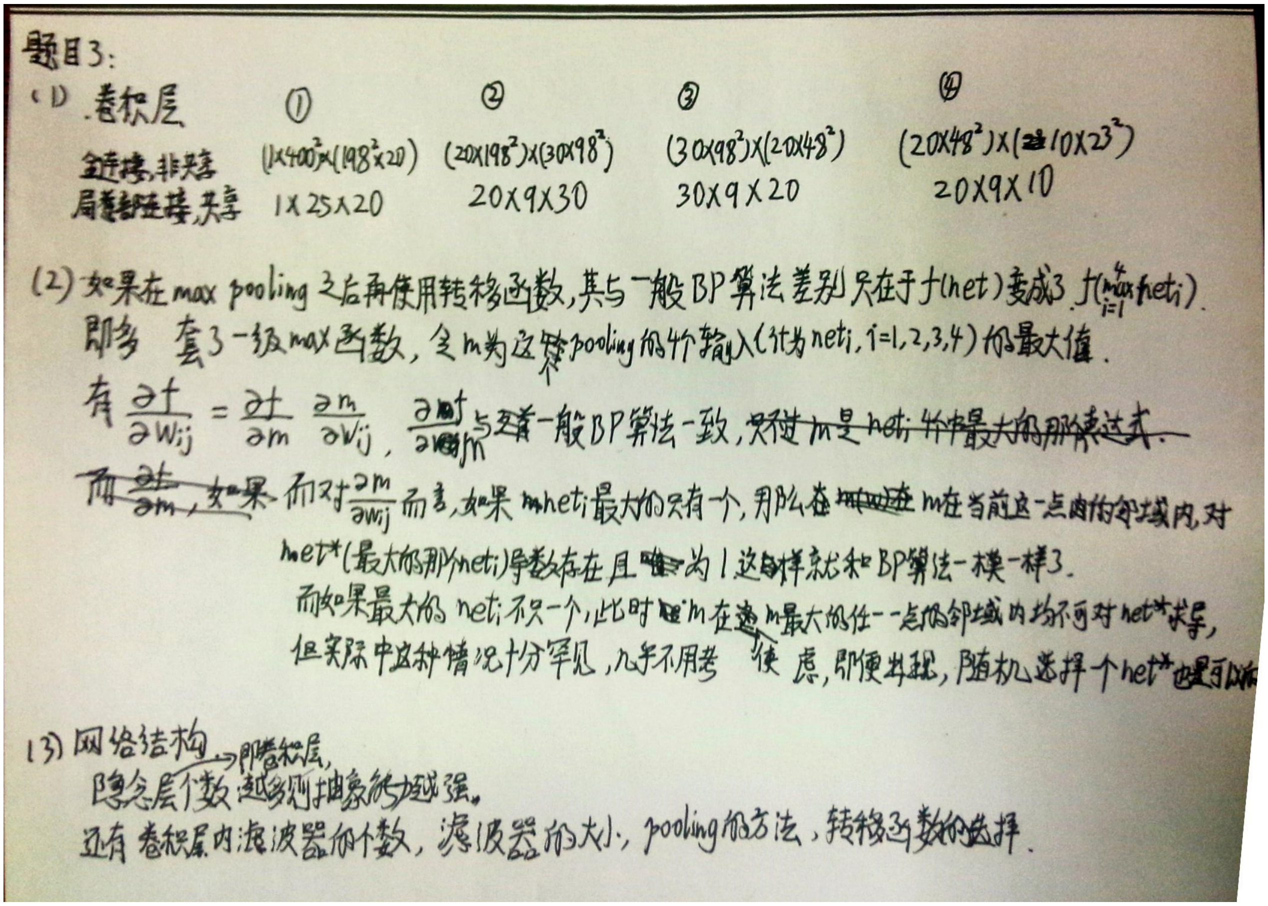
⑦检查是否到结束要求，比如 $\eta(t) < \eta_{min}$.

如是，结束，

否至②.



3 Problem 3



题目3:

(1) 卷积层

	①	②	③	④
全连接,非共享	$(1\times400^2)\times(198^2\times20)$	$(20\times198^2)\times(30\times98^2)$	$(30\times98^2)\times(20\times48^2)$	$(20\times48^2)\times(10\times23^2)$
局部连接,共享	$1\times25\times20$	$20\times9\times30$	$30\times9\times20$	$20\times9\times10$

(2) 如果在max pooling之后再使用转移函数,其与一般BP算法差别只在于$f(net)$变成了$f(\max_{i=1}^{4} net_i)$.
即多套了一级max函数,令m为这个pooling的4个输入(记为net_i, $i=1,2,3,4$)的最大值.
有 $\frac{\partial f}{\partial w_{ij}} = \frac{\partial f}{\partial m}\frac{\partial m}{\partial w_{ij}}$, $\frac{\partial f}{\partial m}$与一般BP算法一致, ~~只需 m是$net_i$中最大的那个表达式.~~
~~而$\frac{\partial f}{\partial m}$,如果~~ 而对$\frac{\partial m}{\partial w_{ij}}$而言,如果$net_i$最大的只有一个,那么m在当前这一点的邻域内,对$net^*$(最大的那个$net_i$)导数存在且为1,这样就和BP算法一模一样了.
而如果最大的net_i不只一个,此时m在使m最大的任一点的邻域内均不可对net^*求导,
但实际中这种情况十分罕见,几乎不用考虑,即便出现,随机选择一个net^*也是可以的

(3) 网络结构→即卷积层,
隐含层个数越多则抽象能力越强。
还有卷积层内滤波器的个数,滤波器的大小,pooling的方法,转移函数的选择.

4 Programming Problem 1

4.1 Result

Question (a) From my experiment, averagely, if the number of nodes in the hidden layer increases, the minimum of judge function will increase, though it requires more passes of training samples. It is difficult to draw a figure to show that trend, since all weights are initially random numbers.

Question (b) Effects of different η have shown in Figures blow. From these two figures we can see that generally, when eta increases, the judging function will decrease to its minimum more quickly.

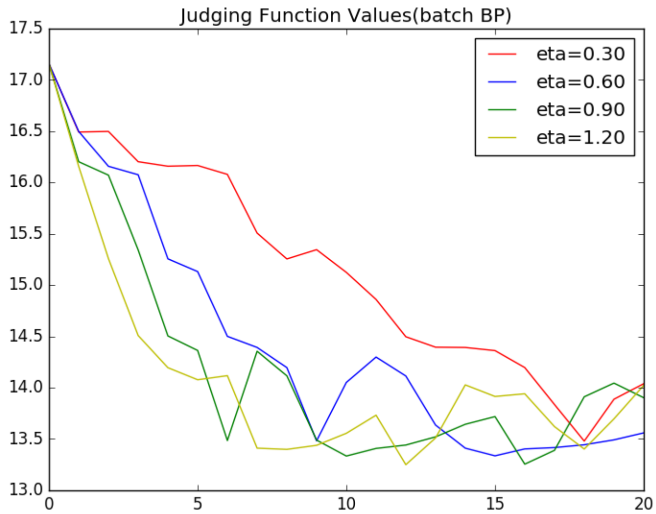


Figure 1: Figure for Batch BP

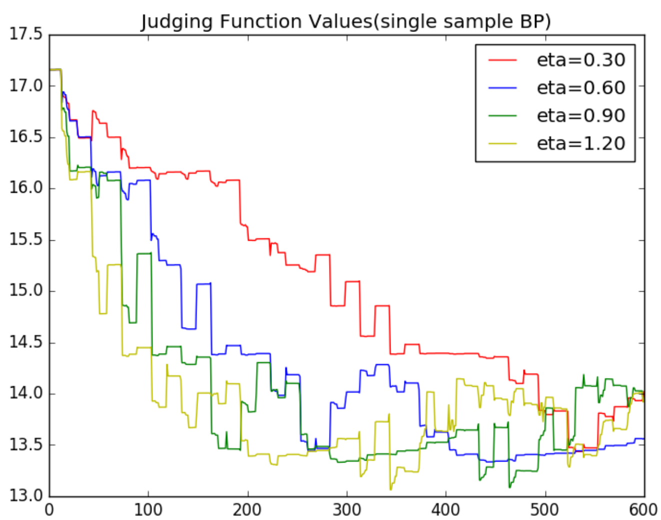


Figure 2: Figure for Single Sample BP

Question (c) Figures are shown above.

4.2 Code

```
#!/usr/bin/python3
# coding=utf-8

import math
import copy
import numpy as np
import matplotlib.pyplot as plt

# Samples
```

```
omega1 = np.array([
    [1.58, 2.32, -5.8],
    [0.67, 1.58, -4.78],
    [1.04, 1.01, -3.63],
    [-1.49, 2.18, -3.39],
    [-0.41, 1.21, -4.73],
    [1.39, 3.16, 2.87],
    [1.20, 1.40, -1.89],
    [-0.92, 1.44, -3.22],
    [0.45, 1.33, -4.38],
    [-0.76, 0.84, -1.96]
])
omega2 = np.array([
    [0.21, 0.03, -2.21],
    [0.37, 0.28, -1.8],
    [0.18, 1.22, 0.16],
    [-0.24, 0.93, -1.01],
    [-1.18, 0.39, -0.39],
    [0.74, 0.96, -1.16],
    [-0.38, 1.94, -0.48],
    [0.02, 0.72, -0.17],
    [0.44, 1.31, -0.14],
    [0.46, 1.49, 0.68]
])
omega3 = np.array([
    [-1.54, 1.71, 0.64],
    [5.41, 3.45, -1.33],
    [1.55, 0.99, 2.69],
    [1.86, 3.19, 1.51],
    [1.68, 1.70, -0.87],
    [3.51, -0.22, -1.39],
    [1.40, -0.44, 0.92],
    [0.44, 0.83, 1.97],
    [0.25, 0.68, -0.99],
    [0.66, -0.45, 0.08]
])


# Constants
sita = 1.5  # When J(w) < sita, training stops.
w_0 = 10  # The initial weight is a uniform U(-w_0, w_0)
eta = 1  # Learning rate
M_bach = 40  # Max training passes of batch BP
M_single = M_bach * 30  # Max training passes of single BP


def trans(layer, x):
    return {
        1: np.tanh(x),  # Hyperbolic function
        2: 1 / (1 + np.exp(-x)),  # sigmoid
        3: x,
    }[layer]


def trans_prime(layer, y):
    return {
```

```
        1: 1 - y**2,  # Hyperbolic function
        2: y * (1 - y),  # sigmoid
        3: 1,
    }[layer]


def init_all_sample():
    s = []
    for x in omega1:
        s.append([x, [1, 0, 0]])
    for x in omega2:
        s.append([x, [0, 1, 0]])
    for x in omega3:
        s.append([x, [0, 0, 1]])
    return s


def init_network(n_h):
    d = 3
    c = 3
    w_ih = np.random.uniform(-w_0, w_0, (d, n_h))
    w_hj = np.random.uniform(-w_0, w_0, (n_h, c))
    w_jo = np.identity(3)  # out
    w_xi = np.identity(3)  # in
    ws = [w_xi, w_ih, w_hj, w_jo]
    # print(ws)
    return ws


def get_net_output(sample, ws):
    out = []
    out.append(sample[0])  # initial x
    for k in range(1, len(ws)):
        w = ws[k]
        y = []
        for j in range(len(w[0])):
            s = 0
            x = out[len(out) - 1]
            for i in range(len(x)):
                s += w[i][j] * x[i]
            s = trans(k, s)
            y.append(s)
        out.append(np.array(y))
    return out


def judge_function(samples, ws):
    s = 0
    for sample in samples:
        out = get_net_output(sample, ws)
        t = sample[1]
        z = out[-1]
        delta = t - z
        s += np.sum([x**2 for x in delta])
    return s / 2
```

```python
def batch_BP(samples, ws, ws_org):
    J = []
    J.append(judge_function(samples, ws))
    for cnt in range(M_bach):
        dws = []
        for w in ws:
            l = (len(w), len(w[0]))
            dws.append(np.zeros(l))
        for k in range(30):
            delta = []
            sample = samples[k % len(samples)]
            t = sample[1]
            out = get_net_output(sample, ws_org)
            z = out[-1]
            delta.append(t - z)
            for layer in range(len(ws) - 2, 0, -1):
                w = ws[layer]
                w_right = ws[layer + 1]
                delta_now = []
                for j in range(len(w[0])):
                    l = [w_right[j][c] * delta[-1][c]
                         for c in range(len(delta[-1]))]
                    sum_right = np.sum(l)
                    delta_now.append(
                        trans_prime(layer, out[layer][j]) * sum_right)
                    for i in range(len(w)):
                        dws[layer][i][j] += eta * \
                            out[layer - 1][i] * delta_now[j]
                delta.append(delta_now)
        for layer in range(len(ws) - 2, 0, -1):
            w = ws[layer]
            for j in range(len(w[0])):
                for i in range(len(w)):
                    w[i][j] += dws[layer][i][j]
        # judge function
        J.append(judge_function(samples, ws))
    return J


def single_sample_BP(samples, ws, ws_org):
    # M_single = 1
    J = []
    J.append(judge_function(samples, ws))
    for k in range(M_single):
        delta = []
        sample = samples[k % len(samples)]
        t = sample[1]
        out = get_net_output(sample, ws_org)
        z = out[-1]
        delta.append(t - z)
        for layer in range(len(ws) - 2, 0, -1):
            w = ws[layer]
            w_right = ws[layer + 1]
            delta_now = []
            for j in range(len(w[0])):
```

```
                l = [w_right[j][c] * delta[-1][c]
                     for c in range(len(delta[-1]))]
                sum_right = np.sum(l)
                delta_now.append(trans_prime(layer, out[layer][j]) * sum_right)
                for i in range(len(w)):
                    dw_ij = eta * out[layer - 1][i] * delta_now[j]
                    w[i][j] += dw_ij
            delta.append(delta_now)
        # judge function
        J.append(judge_function(samples, ws))
    return J

if __name__ == '__main__':
    # ws = init_network(20)
    ws = init_network(4)
    ss = init_all_sample()
    colors = [0,'r','b','g','y']
    for i in range(1,5):
        eta = i * 0.3
        Js_batch = batch_BP(ss, copy.deepcopy(ws), ws)
        label = "eta=" + ('%.2f' % eta)
        plt.plot(Js_batch,color=colors[i], label=label)
    plt.title("Judging Function Values(batch_BP)")
    plt.legend()
    plt.show()

    for i in range(1,5):
        eta = i * 0.3
        Js_batch = single_sample_BP(ss, copy.deepcopy(ws), ws)
        label = "eta=" + ('%.2f' % eta)
        plt.plot(Js_batch,color=colors[i], label=label)
    plt.title("Judging Function Values(single_sample_BP)")
    plt.legend()
    plt.show()
```