人工智能实验：Topic 6——深度学习

Part 1: 深度神经网络

3190102060　黄嘉欣

一、梯度消失

在神经网络中，层数比较多的神经网络模型在训练时有时会出现一些问题，其中比较常见的有梯度消失问题（gradient vanishing problem），其一般随着网络层数的增加会变得越来越明显。

对于如图 1.1 所示的含有 3 个隐藏层的神经网络，当梯度消失问题发生时，接近于输出层的 hidden layer 3 等的权值更新相对正常，但前面的 hidden layer 1 的权值更新会变得很慢，导致前面层的权值几乎不变，仍接近于初始化的权值，这就使得 hidden layer 1 相当于只是一个映射层，对所有的输入做了一个同一映射，此时此深度神经网络的学习等价于只有后几层的浅层网络的学习。

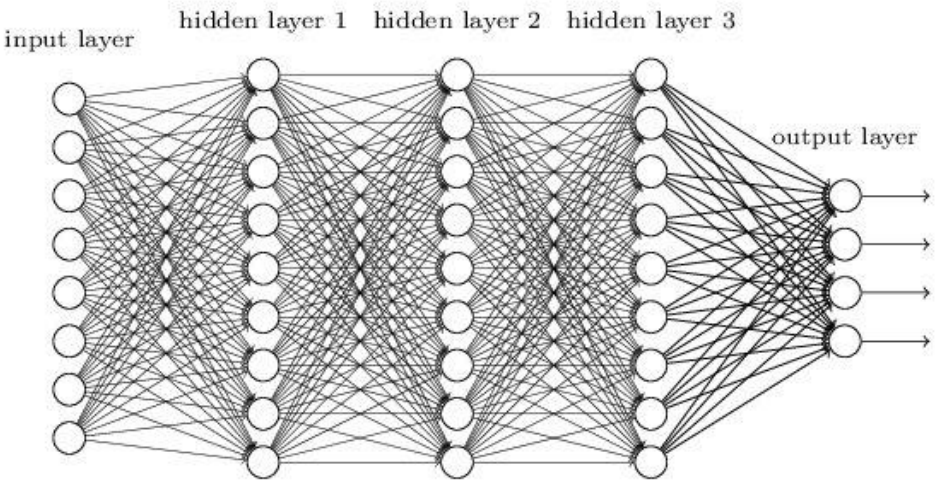


图 1.1 梯度消失问题

为了分析梯度消失问题产生的原因，我们可以分析一个简单的反向传播过程，如下：

图 1.2 反向传播过程

假设每一层只有一个神经元，且每一层输出$y_i = \sigma(z_i) = \sigma(\omega_i x_i + b_i)$，其中$\sigma$为 sigmoid 函数，可以推导出：$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial y_4}\sigma'(z_4)\,\omega_4\sigma'(z_3)\,\omega_3\sigma'(z_2)\,\omega_2\sigma'(z_1)$，而$\sigma'(x)$的最大值为$\frac{1}{4}$，且我们初始化的网络权值$|\omega_i|$通常都小于 1，于是$|\sigma'(z_i)\omega_i| \leq \frac{1}{4}$。因此，对于上面的链式

求导，层数越多，求导结果$\frac{\partial C}{\partial b_1}$越小，从而导致梯度消失的情况出现。

为了解决梯度消失的问题，可以采用 ReLU 激活函数，即：$\varphi(x)=\begin{cases}x, & x>0\\0, & x\le 0\end{cases}=\max(0,x)$，$\varphi'(x)=\begin{cases}1, & x>0\\0, & x\le 0\end{cases}$。

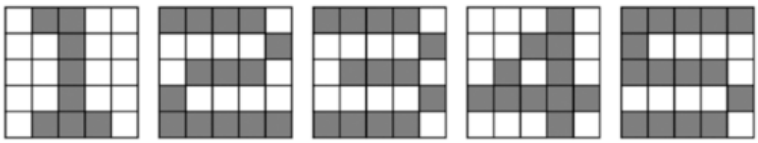
二、过拟合与 Dropout

在深度学习任务当中，过拟合是比较常见的现象，其最直观的表现就是模型在测试集的表现远远差于训练集，也就是模型的泛化性能太差，而这一般是由于训练集大小相对于网络规模偏小。为了缓解过拟合问题，我们可以使用更多、质量更高的数据，也可以减小模型规模，或进行正则化操作等等。常用的 Dropout 技巧包括如下步骤：① 随机（临时）删掉网络中的一部分隐藏神经元，输入输出神经元保持不变；② 输入x通过修改后的网络前向传播，然后把得到的损失结果通过修改的网络反向传播。一小批训练样本执行完这个过程后，在没有被删除的神经元上按照随机梯度下降法更新对应的参数；③ 恢复被删掉的神经元，此时被删除的神经元保持原样，而没有被删除的神经元已经有所更新。不断重复上述过程，直到训练指定轮数，或网络达到阈值要求。

三、实验 6-1

① 实验题目

通过 SGD 训练方法、ReLU 激活函数（最后一层使用 softmax）及 BP 规则，训练深度神经网络，并输出训练后的结果（为简单起见 softmax 函数的导数直接取 1）。训练数据为：



```
X(:, :, 1) = [ 0 1 1 0 0;
               0 0 1 0 0;
               0 0 1 0 0;
               0 0 1 0 0;
               0 1 1 1 0
             ];

X(:, :, 2) = [ 1 1 1 1 0;
               0 0 0 0 1;
               0 1 1 1 0;
               1 0 0 0 0;
               1 1 1 1 1
             ];

X(:, :, 3) = [ 1 1 1 1 0;
               0 0 0 0 1;
               0 1 1 1 0;
               0 0 0 0 1;
               1 1 1 1 0
             ];

X(:, :, 4) = [ 0 0 0 1 0;
               0 0 1 1 0;
               0 1 0 1 0;
               1 1 1 1 1;
               0 0 0 1 0
             ];

X(:, :, 5) = [ 1 1 1 1 1;
               1 0 0 0 0;
               1 1 1 1 0;
               0 0 0 0 1;
               1 1 1 1 0
             ];

D = [ 1 0 0 0 0;
      0 1 0 0 0;
      0 0 1 0 0;
      0 0 0 1 0;
      0 0 0 0 1
    ];
```

② 实验结果与分析

```
实验6-1的训练结果为:
 [[9.99974852e-01 5.85275536e-06 3.23184001e-08 8.51051227e-06
  1.07520832e-05]
 [4.37044310e-07 9.99975893e-01 9.38947280e-06 2.79097448e-08
  1.42523072e-05]
 [1.90090096e-07 1.56682864e-05 9.99977145e-01 6.99690223e-06
  1.71683423e-10]
 [1.16208916e-05 5.98538216e-07 5.22347806e-06 9.99978347e-01
  4.21050729e-06]
 [1.57430155e-05 2.39933450e-07 1.04479703e-05 3.73713121e-08
  9.99973532e-01]]
```

如上，为学习率$\alpha = 0.01$，训练轮数$epochs = 10000$时输出的结果。可以看到，主对角线上的元素均在 0.9997 至 0.9998 之间，与 1 十分接近；其余位置的元素数量级都在10^{-5}及以下，与 0 十分接近。故综合来说，训练后的结果与真值较为吻合，网络设计正确。

四、实验 6-2

① 实验题目

加入 Dropout 技巧，采用 Sigmoid 激活函数（最后一层使用 softmax）及 BP 规则，对实验 6-1 中的深度神经网络进行训练，并输出训练后的结果。

② 实验结果与分析

```
实验6-2的训练结果为:
 [[9.99494478e-01 2.84638421e-04 1.18952625e-04 3.26810848e-05
  6.92496904e-05]
 [8.30747126e-04 9.94987888e-01 4.06421182e-03 1.17037197e-04
  1.15532611e-07]
 [3.51803514e-05 1.42202769e-03 9.97595952e-01 3.13160030e-04
  6.33679669e-04]
 [4.87660723e-05 1.06861731e-05 7.15161839e-04 9.98773356e-01
  4.52029736e-04]
 [2.25361934e-05 1.61303549e-09 1.73465772e-03 1.03173007e-04
  9.98139631e-01]]
```

如上，为学习率$\alpha = 0.01$，训练轮数$epochs = 10000$，Dropout rate=0.2 时所得权重计算出的结果。可以看到，主对角线上的元素均在 0.994 至 0.9995 之间，与 1 较为接近；其余位置元素的数量级都在10^{-3}及以下，与 0 较为接近，故神经网络设计正确。但将此结果与实验 6-1 的输出相比较，可以发现其与真值的误差相对更大。这是由于采用 Dropout 技巧后，训练时网络的规模有所减小，从而影响了结果的准确性；除此之外，

随机初始化的权重可能也会影响最终的训练输出。当然，在面对过拟合的情况时，Dropout 技巧可以有效避免模型过于依赖某些局部特征，进而提高其泛化能力。

五、附录：实验 Python 代码——*DL.py*

```python
import numpy as np

def sigmoid(x):
    """
    sigmoid 函数
    Param:
        x: 输入
    Return:
        y: 对应的 sigmoid 函数值
    """
    y = 1./(1+np.exp(-x))
    return y


def softmax(x):
    """
    softmax 函数
    Param:
        x: 输入
    Return:
        y: 对应的 softmax 函数值
    """
    x = x - np.max(x)
    exp_x = np.exp(x)
    y = exp_x / np.sum(exp_x)
    return y


def ReLU(x):
    """
    ReLU 函数
    Param:
        x: 输入
    Return:
        y: 对应的 ReLU 函数值
    """
    y = np.maximum(0, x)
    return y
```

```python
def DLReLU(X, D, epochs):
    """
    通过 SGD 训练方法、ReLu 激活函数及 BP 规则，构造四层深度学习网络
    Param:
        X: 输入矩阵
        D: Ground Truth
        epochs: 训练轮数
    Return:
        W1, W2, W3, W4: 每层的权重
    """
    # 初始化权重
    W1 = 2*np.random.random((20,25))-1
    W2 = 2*np.random.random((20,20))-1
    W3 = 2*np.random.random((20,20))-1
    W4 = 2*np.random.random((5,20))-1

    alpha = 0.01 # 学习率
    for i in range(epochs):
        for j in range(X.shape[0]):
            input = X[j].reshape(25,1)    # 依次选取输入，25*1

            # forward
            v1 = np.dot(W1, input)        # 第一层相乘结果，20*1
            y1 = ReLU(v1)                 # 第一层输出，20*1

            v2 = np.dot(W2, y1)           # 第二层相乘结果，20*1
            y2 = ReLU(v2)                 # 第二层输出，20*1

            v3 = np.dot(W3, y2)           # 第三层相乘结果，20*1
            y3 = ReLU(v3)                 # 第三层输出，20*1

            v4 = np.dot(W4, y3)           # 第四层相乘结果，5*1
            y4 = softmax(v4)              # 第四层输出，5*1

            # backward
            e4 = D[j].reshape(5,1) - y4   # 第四层输出误差，5*1
            delta4 = e4                   # softmax 的导数取 1
            dW4 = alpha*delta4*y3.T       # 第四层更新值，5*20

            e3 = np.dot(W4.T, delta4)     # 第三层输出误差，20*1
            delta3 = (v3>0)*e3            # 第三层 delta，20*1
            dW3 = alpha*delta3*y2.T       # 第三层更新值，20*20

            e2 = np.dot(W3.T, delta3)     # 第二层输出误差，20*1
```

```
            delta2 = (v2>0)*e2              # 第二层 delta，20*1
            dW2 = alpha*delta2*y1.T         # 第二层更新值，20*20

            e1 = np.dot(W2.T, delta2)       # 第一层输出误差，20*1
            delta1 = (v1>0)*e1              # 第一层 delta，20*1
            dW1 = alpha*delta1*input.T      # 第一层更新值，20*25

            W1 = W1 + dW1                   # 第一层权重
            W2 = W2 + dW2                   # 第二层权重
            W3 = W3 + dW3                   # 第三层权重
            W4 = W4 + dW4                   # 第四层权重
    return W1, W2, W3, W4


def showReLU(X, W1, W2, W3, W4):
    """
    输出 ReLU 训练后的结果
    Param:
        X: 输入矩阵
        W1, W2, W3, W4: 每层的权重
    Return:
    """
    outputs = np.zeros((5,5))
    for i in range(X.shape[0]):
        input = X[i].reshape(25,1)    # 依次选取输入，25*1

        # forward
        v1 = np.dot(W1, input)        # 第一层相乘结果，20*1
        y1 = ReLU(v1)                 # 第一层输出，20*1

        v2 = np.dot(W2, y1)           # 第二层相乘结果，20*1
        y2 = ReLU(v2)                 # 第二层输出，20*1

        v3 = np.dot(W3, y2)           # 第三层相乘结果，20*1
        y3 = ReLU(v3)                 # 第三层输出，20*1

        v4 = np.dot(W4, y3)           # 第四层相乘结果，5*1
        output = softmax(v4)          # 第四层输出，5*1

        outputs[i,:] = output.reshape(1,5) # 保存结果
    print('实验 6-1 的训练结果为: \n', outputs, '\n')


def Dropout(y, p):
    """
```

```
    Dropout 技巧
    Param:
        y: 输入向量
        p: Dropout rate
    Return:
        ym: 激活值
    """
    ym = np.zeros_like(y)              # 与 y 同样维度，全 0
    n_remain = round(len(y)*(1-p))     # 保留的神经元数量
    idxs = np.random.choice(len(y), n_remain, False) # 保留的神经元索引
    ym[idxs] = 1/(1-p)                 # 保留的神经元乘以 1/(1-p)
    return ym


def DLDropout(X, D, epochs):
    """
    加入 Dropout 技巧，构造四层深度学习网络
    Param:
        X: 输入矩阵
        D: Ground Truth
        epochs: 训练轮数
    Return:
        W1, W2, W3, W4: 每层的权重
    """
    # 初始化权重
    W1 = 2*np.random.random((20,25))-1
    W2 = 2*np.random.random((20,20))-1
    W3 = 2*np.random.random((20,20))-1
    W4 = 2*np.random.random((5,20))-1

    alpha = 0.01 # 学习率
    p = 0.2      # Dropout ratio
    for i in range(epochs):
        for j in range(X.shape[0]):
            input = X[j].reshape(25,1)    # 依次选取输入，25*1

            # forward
            v1 = np.dot(W1, input)        # 第一层相乘结果，20*1
            y1 = sigmoid(v1)              # 第一层输出，20*1
            y1 = y1*Dropout(y1, p)        # Dropout

            v2 = np.dot(W2, y1)           # 第二层相乘结果，20*1
            y2 = sigmoid(v2)              # 第二层输出，20*1
            y2 = y2*Dropout(y2, p)        # Dropout
```

```
            v3 = np.dot(W3, y2)             # 第三层相乘结果，20*1
            y3 = sigmoid(v3)                # 第三层输出，20*1
            y3 = y3*Dropout(y3, p)          # Dropout

            v4 = np.dot(W4, y3)             # 第四层相乘结果，5*1
            y4 = softmax(v4)                # 第四层输出，5*1

            # backward
            e4 = D[j].reshape(5,1) - y4     # 第四层输出误差，5*1
            delta4 = e4                     # softmax 的导数取 1
            dW4 = alpha*delta4*y3.T         # 第四层更新值，5*20

            e3 = np.dot(W4.T, delta4)       # 第三层输出误差，20*1
            delta3 = y3*(1-y3)*e3           # 第三层 delta，20*1
            dW3 = alpha*delta3*y2.T         # 第三层更新值，20*20

            e2 = np.dot(W3.T, delta3)       # 第二层输出误差，20*1
            delta2 = y2*(1-y2)*e2           # 第二层 delta，20*1
            dW2 = alpha*delta2*y1.T         # 第二层更新值，20*20

            e1 = np.dot(W2.T, delta2)       # 第一层输出误差，20*1
            delta1 = y1*(1-y1)*e1           # 第一层 delta，20*1
            dW1 = alpha*delta1*input.T      # 第一层更新值，20*25

            W1 = W1 + dW1                   # 第一层权重
            W2 = W2 + dW2                   # 第二层权重
            W3 = W3 + dW3                   # 第三层权重
            W4 = W4 + dW4                   # 第四层权重
    return W1, W2, W3, W4


def showDropout(X, W1, W2, W3, W4):
    """
    加入 Dropout 技巧训练后的结果
    Param:
        X: 输入矩阵
        W1, W2, W3, W4: 每层的权重
    Return:
    """
    outputs = np.zeros((5,5))
    for i in range(X.shape[0]):
        input = X[i].reshape(25,1)    # 依次选取输入，25*1

        # forward
        v1 = np.dot(W1, input)        # 第一层相乘结果，20*1
```

```
        y1 = sigmoid(v1)              # 第一层输出，20*1

        v2 = np.dot(W2, y1)           # 第二层相乘结果，20*1
        y2 = sigmoid(v2)              # 第二层输出，20*1

        v3 = np.dot(W3, y2)           # 第三层相乘结果，20*1
        y3 = sigmoid(v3)              # 第三层输出，20*1

        v4 = np.dot(W4, y3)           # 第四层相乘结果，5*1
        output = softmax(v4)          # 第四层输出，5*1

        outputs[i,:] = output.reshape(1,5) # 保存结果
    print('实验 6-2 的训练结果为: \n', outputs)


def main():
    # 初始化 X 和 D
    X = np.array([[0,1,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,1,1,1,0],
                  [1,1,1,1,0,0,0,0,0,1,0,1,1,1,0,1,0,0,0,0,1,1,1,1,1],
                  [1,1,1,1,0,0,0,0,0,1,0,1,1,1,0,0,0,0,0,1,1,1,1,1,0],
                  [0,0,0,1,0,0,0,1,1,0,0,1,0,1,0,1,1,1,1,1,0,0,0,1,0],
                  [1,1,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0,1,1,1,1,1,0]])
    D = np.array([[1,0,0,0,0],
                  [0,1,0,0,0],
                  [0,0,1,0,0],
                  [0,0,0,1,0],
                  [0,0,0,0,1]])

    epochs = 10000                          # 训练轮数

    # lab 6-1
    W1, W2, W3, W4 = DLReLU(X, D, epochs)  # 训练权重
    showReLU(X, W1, W2, W3, W4)            # 输出训练后的结果

    # lab 6-2
    W1, W2, W3, W4 = DLDropout(X, D, epochs) # 训练权重
    showDropout(X, W1, W2, W3, W4)           # 输出训练后的结果


if __name__ == '__main__':
    main()
```