中山大学数据科学与计算机学院
移动信息工程专业—人工智能
本科生实验报告

（2017-2018 学年秋季学期）

课程名称：**Artificial Intelligence**

教学班级	1506	专业（方向）	移动信息工程（互联网）
学号	15352116	姓名	洪子淇

1 实验题目

1. 实现三层神经网络（输入层，隐藏层，输出层）
2. 对神经网络进行优化，例如数据预处理，加入正则项等
3. 理解神经网络的激活函数，思考梯度消失和梯度爆炸产生的原因

2 实验内容

2.1 算法原理

2.1.1 数据预处理

舍弃某些特征

对于这个系统来说，样例的 id 是没有意义的，所以不考虑把该特征加入神经网络中，而对于日期，由于可以有年月日得到，所以也考虑不把该特征加入作为输入层的输入

标准化（加深理解后）

常用标准化的方法有：最大-最小标准化，z-score 标准化等。

- 最大-最小标准化

  最大-最小标准化就是将原始数据进行线性变化之后映射到 [0,1] 区间，假设 X_{min},X_{max} 分别为原始数据的最小值，最大值，公式如下：

  $$\hat{X} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

  缺点：当存在极端大值或极端小值的时候，就会把原来的数据集压缩到一个比较小的范围

- z-score 标准化

  z-score 标准化也是对原始数据进行线性变化，具体含义表示原始数据和平均数之间的距离，距离是以标准差为单位计算的。当原始数据出现极端大值或极端小值的时候也不会对数据有很大的影响，公式如下：

  $$\hat{X} = \frac{X - \mu}{\sigma}$$

由于给定的数据集中连续变化的气温，体感气温，湿度以及风力已经进行标准化处理。所以本次实验不需要对他们进行标准化处理。

这里主要是分析标准化处理之后的好处：

1. 将有量纲的数据转换为无量纲，消除特征之间的量纲影响。
2. 提升模型的收敛速度
3. 具有正则化的效果

上面是基于对所有的特征梯度下降设置同一步长而言的。假设有两个特征值 x_1,x_2, 那么预测函数为 $\hat{y} = w_0 + w_1x_1 + w_2x_2$, 损失函数为 $J = (\hat{y} - y)^2$

$$\begin{cases} \dfrac{\partial J}{\partial w_1} = 2x_1(\hat{y} - y) \\ \dfrac{\partial J}{\partial w_2} = 2x_2(\hat{y} - y) \end{cases}$$

假设 x_1 远远小于 x_2 的时候，在计算距离公式 x_1 几乎可以被忽略了。从正则化系数考虑，如果没有标准化，可能导致系数间的比例与数据间的比例呈现反比例关系，那么可能导致模型难以收敛。

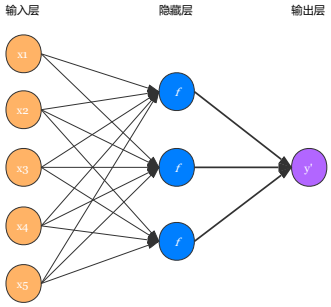
假设 x_1 小于 x_2 的时候，那么迭代求解的过程就会比较曲折，影响收敛速度；标准化之后寻找最优过程比较平缓，也更容易收敛到最优解，下面是标准化前后寻优过程的对比图[1]：

特征二进制化

本实验针对 season, mnth, hr, weekday, weathersit 特征属性进行处理，由于对于这类特征来说，如果使用离散的数值表示不同的属性，就会产生同一个特征下不同的属性之间有距离上的差距，而实际上属性之间是平等的。为了消除同一特征下属性之间的距离差距，使用二进制编码方式表示这个特征。例如，对于特征 season$\{1, 2, 3, 4\}$，采用四列特征 ('season_1', 'season_2','season_3','season_4') 共同表示：$\{1000, 0100, 0010, 0001\}$。

2.1.2 BPNN

本实验中，BPNN 包括了前馈神经网络 (前向传递阶段) 以及误差逆传播（反向传递阶段）两个过程。(以三层感知器为例，拓扑结构如图下所示)



- 前向传递阶段

  输入层乘上第 i 个连接权重向量输入到隐藏层的第 i 个神经元节点，隐藏层激活之后乘上连接权重向量得到预测 y 值。常用的激活函数有 sigmoid，tanh，ReLu 等。他们之间的差别见思考题的分析。

- 反向传递阶段

  通过输出层输出的 y 值计算损失函数，应用 BGD,SGD 等算法求损失函数的最小值，对权重向量进行更新。

损失函数以及正则项

本实验中取输出层节点的误差平方和作为损害函数，即 $MSE = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$

损失函数加入正则项可以提高模型的泛化能力，防止过拟合现象：

当特征过多，会导致我们的模型复杂度上升，容易产生过拟合现象 (对训练集误差较小，对测试集的误差 query 比较大)，为了防止过拟合，可以适当减少一些不重要的特征参数，而这可以通过规则函数来实现。规则函数可以是模型向量的范数，一般有零范数，一范数，二范数等。零范数表示向量中不为零的个数，当零范数比较小的时候，很直观得出特征参数也比较小，从而提高了模型的泛化能力，由于难以对零范数进行优化求解，所以机器学习会采取添加一范数或者二范数来提高模型的泛化能力。这里简单总结一下一范数和二范数之间的区别：

- 一范数是零范数的最优凸近似，可以实现稀疏化，即将某些不重要的特征对应权重重置为 0，从而提高模型的泛化能力。
- 二范数可以使得权重向量的每个元素都很小，接近于 0。当参数越小，也就是减少了某些特征分量的影响性，越不容易产生过拟合现象。(粗略地说，大的权重值会使得特征上的变动引起 wx 大的变动，因此需要避免[2]) 而且二范数比起一范数更容易优化求解，本实验中的规则函数便是二范数。所以新的损失函数为：

$$J = \frac{1}{2m}\sum_{t=1}^{m}(\hat{y}_t - y_t)^2 + \frac{\lambda}{2}\sum_{l=1}^{nl-1}\sum_{i=1}^{S_l}\sum_{j=1}^{S_{l+1}}\left(w_{ji}^{l}\right)^2$$

其中，w_{ji}^{l} 表示第 l 层的连接权重向量，$\hat{y}_t$ 表示第 t 个样本的预测值，即输出层的输出值。假设激活函数为 $f(h_j)$，其中 h_j 为隐藏层第 j 个神经节点的输入（输出层与第 j 个连接权重向量乘积的线性组合）：

$$\hat{y}_t = \sum_{j=1}^{S_{l+1}} w_j^{l=2} f(h_j) + b^{l=2}\ ,\quad h_j = \sum_{i=1}^{s_l} w_{ji}^{l=1} x_i + b_j^{l=1}$$

由于损失函数是一个凸函数，对权重向量求偏导数，得到当前权重向量下降方向，然后根据下降方向设置步长，可以不断逼近最优解。

$$\therefore \begin{cases} \dfrac{\partial J}{\partial w_j^{l=2}} = \dfrac{1}{m}\sum_{t=1}^{m} w_j^{l=2}(\hat{y}_t - y_t) f(h_j) + \lambda w_j^{l=2} \\ \dfrac{\partial J}{\partial b^{l=2}} = \dfrac{1}{m}\sum_{t=1}^{m} w_j^{l=2}(\hat{y}_t - y_t) + \lambda b^{l=2} \end{cases}$$

$$\begin{cases} \dfrac{\partial J}{\partial w_{ji}^{l=1}} = \dfrac{1}{m}\sum_{t=1}^{m} w_j^{l=2}(\hat{y}_t - y_t)\dfrac{\partial f(h_j)}{\partial h_j}\dfrac{\partial h_j}{\partial w_{ji}^{l=1}} + \lambda w_{ji}^{l=1} \\ \dfrac{\partial J}{\partial b_j^{l=1}} = \dfrac{1}{m}\sum_{t=1}^{m} w_j^{l=2}(\hat{y}_t - y_t)\dfrac{\partial f(h_j)}{\partial h_j}\dfrac{\partial h_j}{\partial b_j^{l=1}} + \lambda b_j^{l=1} \end{cases}$$

假设 sigmoid 作为激活函数，便有 $\partial f(h_j)/\partial h_j = f(h_j)(1-f(h_j))$（ps：对于其他激活函数，只需要对公式中的 $\partial f(h_j)/\partial h_j$ 部分进行修改）。

$$\begin{cases} \dfrac{\partial J}{\partial w_{ji}^{l=1}} = \dfrac{1}{m}\sum_{t=1}^{m} w_j^{l=2}(\hat{y}_t - y_t)f(h_j)(1-f(h_j))x_i + \lambda w_{ji}^{l=1} \\ \dfrac{\partial J}{\partial b_j^{l=1}} = \dfrac{1}{m}\sum_{t=1}^{m} w_j^{l=2}(\hat{y}_t - y_t)f(h_j)(1-f(h_j)) + \lambda b_j^{l=1} \end{cases}$$

一般有以下几种逼近最优解的方法：BGD, SGD, MBGD 等。

BGD

BGD，全称为 Batch Gradient Descen，t 也叫批量梯度下降法。每次迭代更新权重向量：使用全部的样本数据，计算开始的权重向量对整体样本的损失程度，然后更新权重向量，当样本个数比较大的时候，更新权重向量的速度比较慢，当步长越小，收敛速度也越慢。可以根据上面的偏导数乘上学习率 α，计算权重向量的更新公式。

SGD

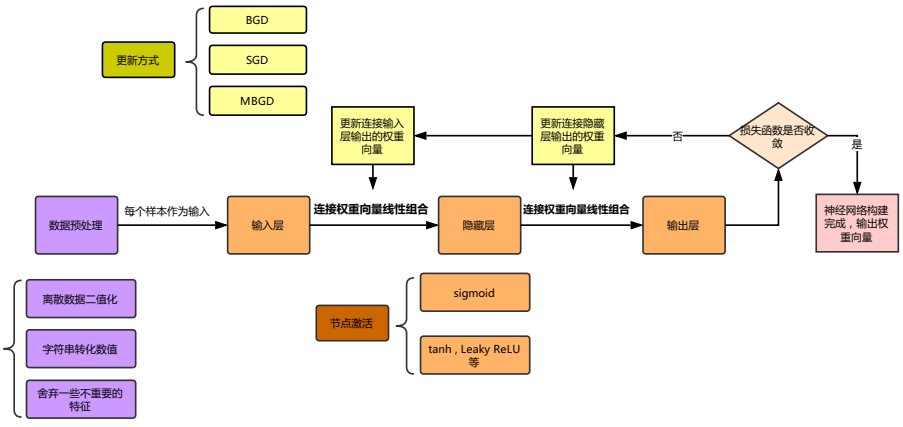
SGD，全称 stochastic Gradient Descent，也叫随机梯度下降。和 BGD 最大不同就是每次通过一个样本对权重向量进行更新，带来的优点就是训练速度快，可是由于每次只考虑了单个样本，所以下降方向的准确率也会有所下降，导致迭代次数增加。更新公式为：

$$\begin{cases} w_j^{l=2} = w_j^{l=2} - \alpha\left(w_j^{l=2}(\hat{y}_t - y_t)f(h_j) + \lambda w_j^{l=2}\right) \\ b^{l=2} = b^{l=2} - \alpha\left(w_j^{l=2}\hat{y}_t - y_t + \lambda b^{l=2}\right) \\ w_{ji}^{l=1} = w_{ji}^{l=1} - \alpha\left(w_j^{l=2}(\hat{y}_t - y_t)f(h_j)(1-f(h_j))x_i + \lambda w_{ji}^{l=1}\right) \\ b_j^{l=1} = b_j^{l=1} - \alpha\left(w_j^{l=2}(\hat{y}_t - y_t)f(h_j)(1-f(h_j)) + \lambda b_j^{l=1}\right) \end{cases}$$

MBGD

MBGD 是基于 BGD 和 SGD 的折中，每次选取 k 个训练样本进行迭代，每次迭代更新权重向量考虑 k 个训练样本。和 BGD 更新权重向量的公式一样，只不过把全体样本数 m，转换为 k($k < m$)。

神经网络模型框图



本实验中验证集选择训练集的最后 475 个。

2.2 伪代码

数据预处理

Algorithm 1 Data Preparation

Require: Input database
Ensure: Output new database
feature = ['season', 'weathersit', 'weekday', 'mnth', 'hr']
Update feature values into binary code
Drop 'instant', 'dteday' from database

BPNN

Algorithm 2 BPNN

function BilayerRModel(train, hwidth)
    Initialize W_0, b_0 randomly
    α = 10^{-4} ▷ α means learning rate
    eps = 10^{-10}
    λ = 1 ▷ λ means regularization coefficient
    J_0 = MSE(train, train_y, W_0, b_0) + RegularTerms
    loop t times
        switch (Method in GD)
            case BGD**:**
                $W = W_0 - \frac{\alpha}{m}\sum_{i=1}^{m}\nabla W_0 - \alpha\lambda W_0$
                $b = b_0 - \frac{\alpha}{m}\sum_{i=1}^{m}\nabla b_0 - \alpha\lambda b_0$
            end case
            case SGD**:**
                loop m times ▷ m means train samples' number
                    $W = W_0 - \alpha\nabla W_0 - \alpha\lambda W_0$
                    $b = b_0 - \alpha\nabla b_0 - \alpha\lambda b_0$
                end loop
            end case
            case MBGD**:**
                select k samples from train randomly
                loop k times
                    $W = W_0 - \frac{\alpha}{k}\sum_{i=1}^{k}\nabla W_0 - \alpha\lambda W_0$
                    $b = b_0 - \frac{\alpha}{k}\sum_{i=1}^{k}\nabla b_0 - \alpha\lambda b_0$
                end loop
            end case
        end switch
    end loop
end function

Algorithm 3 BPNN

function Activate(x)
    return $1/(1+e^{-x})$ ▷ Sigmoid function
end function

function Predict(test, W, b)
    $h_j = \sum w_{ji}^{l=1} + b_j^{l=1}$ ▷ $w_{ji}^{l=1}$ is weight connecting x_i with jth Hidden Node
    ▷ $w_j^{l=2}$ is weight connecting jth Hidden Node with Output Node
    return $\sum w_j^{l=2} Activate(h_j) + b^{l=2}$
end function

function MSE(train, train_y, W, b)
    $\hat{Y} = Predict(train)$
    return $\frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - train_{y_i})^2$ ▷ $\hat{y}_i \, \epsilon \, \hat{Y}$, $train_{y_i} \, \epsilon \, train_y$
end function

2.3 关键代码截图

数据预处理

```
season = pd.get_dummies(train_df['season'],prefix='season')
weathersit = pd.get_dummies(train_df['weathersit'],prefix='weathersit')
weekday = pd.get_dummies(train_df['weekday'],prefix='weekday')
mnth = pd.get_dummies(train_df['mnth'],prefix='mnth')
hr = pd.get_dummies(train_df['hr'],prefix='hr')

train_df = pd.concat([train_df,season],axis=1)
train_df = pd.concat([train_df,weathersit],axis=1)
train_df = pd.concat([train_df,weekday],axis=1)
train_df = pd.concat([train_df,mnth],axis=1)
train_df = pd.concat([train_df,hr],axis=1)
```

多维表示

拼接

BSD

每次遍历整个文本的时候，每个样本使用遍历前的权重向量。

```
        nr = np.random.rand
        self.w = [nr(xdim, hwidth), nr(hwidth, 1)]
        self.b = [nr(1, hwidth), nr(1, 1)]
        wc, bc = copy.deepcopy(self.w), copy.deepcopy(self.b)    # copy
        self.losses = {'train':[], 'validation': []}
        t_mse, v_mse = self.sse(train_X, train_y), self.sse(vali_X, vali_y)
        self.losses['train'] += [t_mse]
        self.losses['validation'] += [v_mse]
        k = math.ceil(m)
        for t in range(times):
            for i in range(m):
                o1 = sigmoid(np.dot(train_X[[i]], wc[0]) + bc[0]) # (1, hwidth) # use the w out of loop i
                o2 = np.dot(o1, wc[1]) + bc[1]
                g = o2 - train_y[i] # (1, 1)
                e = o1 * (1 - o1) * self.w[1].T * g # (1, hwidth)
                self.w[1] -= alpha * g * o1.T / k
                self.b[1] -= alpha * g / k
                self.w[0] -= alpha * np.dot(train_X[[i]].T, e) / k
                self.b[0] -= alpha * e / k
            self.w[1] -= alpha*lam * self.w[1] #regular terms
            self.b[1] -= alpha*lam * self.b[1]
            self.w[0] -= alpha*lam * self.w[0]
            self.b[0] -= alpha*lam * self.b[0]
            t_mse, v_mse = self.sse(train_X, train_y), self.sse(vali_X, vali_y)
            self.losses['train'] += [t_mse]
            self.losses['validation'] += [v_mse]
            J = t_mse + lam * self.rd()
            print('times:', t, ' mse:', t_mse,' J:', J)
```

SGD

每次遍历一个样本的时候，就对权重进行更新，也即下一次样本使用上一次样本更新的权重向量进行预测。

```
        self.w = [nr(xdim, hwidth), nr(hwidth, 1)]
        self.b = [nr(1, hwidth), nr(1, 1)]
        self.losses = {'train':[], 'validation': []}
        t_mse, v_mse = self.sse(train_X, train_y), self.sse(vali_X, vali_y)
        self.losses['train'] += [t_mse]
        self.losses['validation'] += [v_mse]
        for t in range(times):
            for i in  random.sample(range(m), m):
                o1 = sigmoid(np.dot(train_X[[i]], self.w[0]) + self.b[0]) # (1, hwidth)
                o2 = np.dot(o1, self.w[1]) + self.b[1] # (1, 1)
                g = o2 - train_y[i] # (1, 1)
                e = o1 * (1 - o1) * self.w[1].T * g # (1, hwidth)
                self.w[1] = (1 - alpha*lam) * self.w[1] - alpha * g * o1.T # change w from last w in loop_i
                self.b[1] = (1 - alpha*lam) * self.b[1] - alpha * g
                self.w[0] = (1 - alpha*lam) * self.w[0] - alpha * np.dot(train_X[[i]].T, e)
                self.b[0] = (1 - alpha*lam) * self.b[0] - alpha * e
            t_mse, v_mse = self.sse(train_X, train_y), self.sse(vali_X, vali_y)
            self.losses['train'] += [t_mse]
            self.losses['validation'] += [v_mse]
            J = t_mse + lam * self.rd()
            print('times:', t, ' mse:', t_mse,' J:', J)
        pass
```

MBGD

每次随机取出一部分的样本进行遍历，每遍历一个样本就对权重向量进更新，也就是 miniSGD。

```
        nr = np.random.rand
        self.w = [nr(xdim, hwidth), nr(hwidth, 1)]
        self.b = [nr(1, hwidth), nr(1, 1)]
        self.losses = {'train':[], 'validation': []}
        t_mse, v_mse = self.sse(train_X, train_y), self.sse(vali_X, vali_y)
        self.losses['train'] += [t_mse]
        self.losses['validation'] += [v_mse]
        k = math.ceil(0.577 * m) # select part of train set
        for t in range(times):
            for i in random.sample(range(m), k):    #suffle the train samples
                o1 = sigmoid(np.dot(train_X[[i]], self.w[0]) + self.b[0]) # (1, hwidth)
                o2 = np.dot(o1, self.w[1]) + self.b[1] # (1, 1)
                g = o2 - train_y[i] # (1, 1)
                e = o1 * (1 - o1) * self.w[1].T * g # (1, hwidth)
                self.w[1] = (1 - alpha*lam) * self.w[1] - alpha * g * o1.T # change w from last w in loop_i
                self.b[1] = (1 - alpha*lam) * self.b[1] - alpha * g
                self.w[0] = (1 - alpha*lam) * self.w[0] - alpha * np.dot(train_X[[i]].T, e)
                self.b[0] = (1 - alpha*lam) * self.b[0] - alpha * e
            t_mse, v_mse = self.sse(train_X, train_y), self.sse(vali_X, vali_y)
            self.losses['train'] += [t_mse]
            self.losses['validation'] += [v_mse]
            J = t_mse + lam * self.rd()
            print('times:', t, ' mse:', t_mse,' J:', J)
        pass
```

2.4 创新点 & 优化

1. 实现了随机梯度下降以及小批量梯度下降更新权重；
2. 损失函数加入正则项
3. 对数据进行预处理：删除一些冗余特征属性（比如$'dtday','instant'$）；对离散性变量（如$'season','mnth','hr'$等）用 0,1 编码表示，实现属性之间的等距化

3 实验结果及分析

3.1 实验结果展示实例

小数据：

$season$	yr	$mnth$	hr	cnt
1	0	1	0	16

权重数据（after 表示遍历一次更新的权重，其中隐藏层节点数为 1，步长为 1）：

	w_0				b_0	w_1	b_1
before	0.03642065	0.62494038	0.91966806	0.924233	0.64037687	0.27040329	0.14754273
after	0.62841526	0.62494038	1.51166267	0.924233	1.23237148	13.2651337	15.77515322

验算过程：

前向传播：

隐藏层节点输出：$o1 = sigmoid(\sum_1^4 w_0^i x^i + b_0) = 0.83152382$

输出层节点输出：$o2 = w_1 o_1 + b_1 = 0.37238951$

后向传播：

w_1 更新：$w_1 = w_1 - (o_2 - 0)o_1 = 13.2651337$

b_1 更新：$b_1 = b_1 - (o_2 - 0) = 15.77515322$

w_0 更新：$w_0^i = w_0 - w_1(o_2 - 0)o_1(1 - o_1)x^i$

$w_0 = [0.62841526 \quad 0.62494038 \quad 1.51166267 \quad 0.924233]$

b_0 更新：$b_0 = b_0 - w_1(o_2 - 0)o_1(1 - o_1) = 1.23237148$

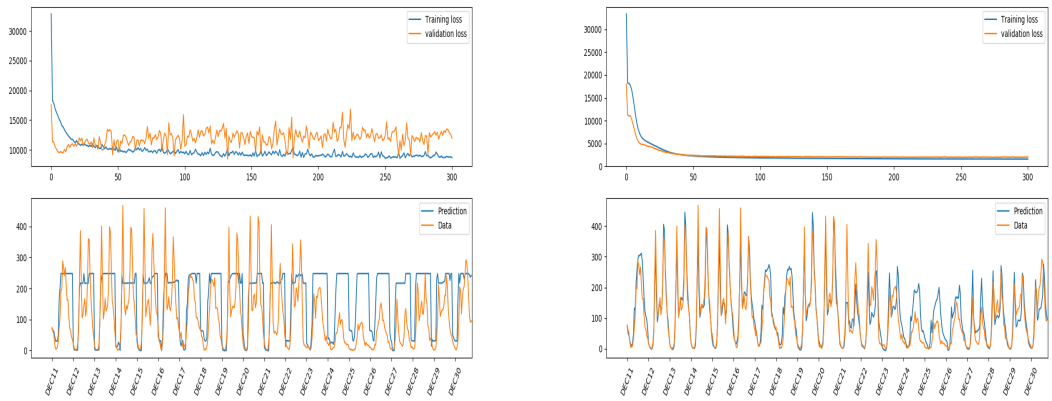
实验结果如下：

```
Backwark Pass:
w[0]: [[ 0.62841526  0.62494038  1.51166267  0.924233  ]] b[0]: [[ 1.23237148]]
w[1]: [[ 13.2651337]] b[1]: [[ 15.77515322]]
```

3.2 评测指标展示即分析

SGD

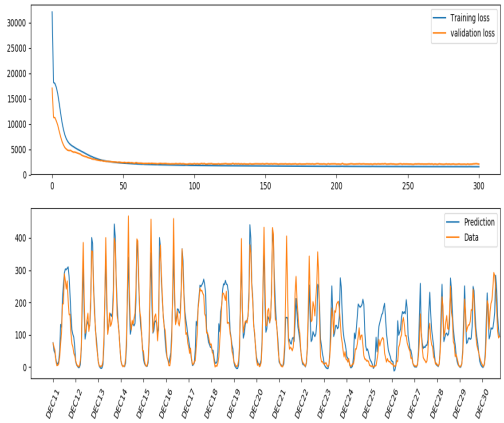
设置步长为 10^{-5}，迭代次数为 300：原始的 SGD 的实验结果与优化后的 SGD 的比较如图下所示：原始指的是没有对特征属性进行等距化处理，没有加入正则项。优化 SGD 正则项常数设置为 0.1



实验结果分析：原始的 SGD 在迭代 300 次的时候 MSE 大概降到 8000 左右，优化后的 SGD 可以降到 1500 左右。证明数据预处理可以加快神经网络的收敛速度，更快地建立模型。

MSGD

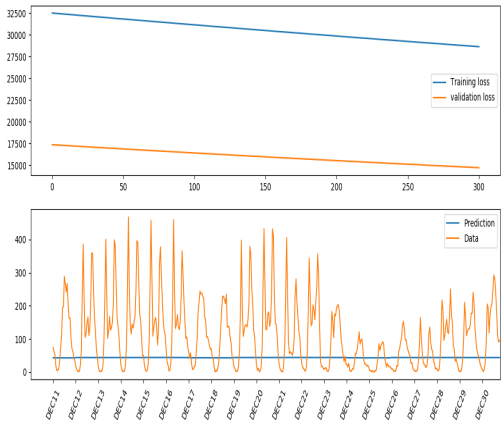
下面是设置步长为 10^{-6} 同样迭代次数为 300 的 MSGD(进行了预处理以及正则化) 的实验结果:



实验结果分析: MSGD 与 SGD 更新权重的方式一样，不过减少了训练集的样本个数。为了防止样本个数较少带来的过拟合，所以每次遍历的时候从全体样本中随机选择 k 份。从结果可以看出和 SGD 效果差不多，由于遍历样本数目减少，所以每次迭代时间减少。

BGD

下面是设置步长为 10^{-5} 同样迭代次数为 300 的 BGD(进行了预处理以及正则化) 的实验结果:

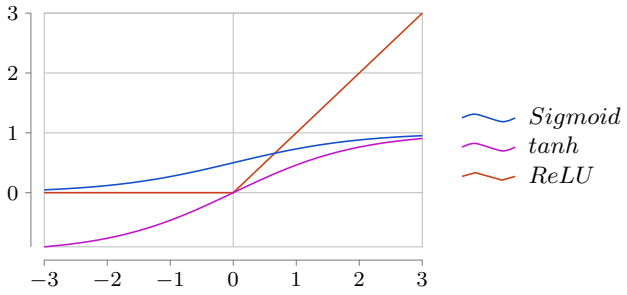


实验结果分析: 由于每次迭代的时候使用的是最初的权重向量，所以每一次样本之间的更新可能会抵消掉，所以更新权重效果不明显。对比 SGD 每一个样本都考虑了上一个样本的损失程度，在上一个样本的损失下对权重向量进行更新，由于是每一个样本更新一次，所以梯度下降曲线比较曲折。对比 BGD，可能更加合理。至于对 BGD 的优化方向考虑了样本随机性，就是 MBGD。或者对样本进行聚类。以此缓解样本之间梯度的抵消。(这里只是提出想法，没有具体实现)

4 思考题

1. 说明下其他激活函数的优缺点 (激活函数：sigmoid，tanh，Relu)

   首先这些激活函数都是非线性函数，对于神经网络来说，可以有效对非线性数据进行建模（在处理分类问题中，如果激活函数是线性的，而且连接函数也是线性的，就是说深井网络中只存在着线性运算的时候，那么该网络的结果也只是一个线性映射）。由于这三个函数是非线性的，他们都具有非线性映射的学习能力[3]。



   (a) Sigmoid

$$f(x) = \frac{1}{1+e^{-x}}$$

   缺点：对于 $[-5,5]$ 之间才有比较好的激活性，当在这个范围外的时候，两侧的梯度比较平缓而且趋近 0（也叫正负饱和区）。而且当它用做中间的隐层的激活层的时候，由于求导会向上一层传递 $f(x)'$，$(f(x)(1-f(x)))$。通过观测 Sigmoid 的函数曲线，一旦 x 落入了左右饱和区域，那么 $f(x)'$ 接近于 0，所以向上一层传递的梯度也非常小，链式求导法则会使得上一层的梯度也非常小，就会使得网络权重向量难以得到有效训练（梯度消失现象）。

   (b) Tanh

$$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

   优点：梯度是 Sigmoid 函数梯度的两倍，所以收敛速度会比 Sigmoid 要快。

   缺点：Tanh 通过 sigmoid 的线性变化得到，所以不能改善函数两端过于平坦的现象，依旧对大值敏感度不高，具有 Sigmoid 一样的缺点，也会产生梯度消失的现象。

   (c) ReLU

   原始的 ReLU 公式为：

$$\begin{cases} y = x & x > 0 \\ y = 0 & x \leq 0 \end{cases}$$

   优点：ReLU 是对上面两个非线性函数的在右饱和区域的改善。由于 $x > 0$ 的时候导数为 1，所以 ReLU 可以在 $x > 0$ 可以缓解梯度消失的问题。

   缺点：ReLU 有一个很明显的缺点，当 $x < 0$ 的时候，梯度为 0，那么就会导致所有权重不更新（神经元死亡）。对于 ReLU 的改善有 Leaky ReLU，在 $x < 0$ 的时候 $y = ax$, 可以消除神经元死亡的现象。

2. 有什么方法可以实现传递过程中不激活所有节点?

   可以在激活前加多一个判定条件，例如对 Sigmoid 函数来说，对于输入值为较大的值的时候，输出值将近 0，对于激活这个函数之后再乘上对应的权重依旧也是一个近乎 0 的数值，对于输出层的影响不大，所以可以考虑不激活这个节点。

3. 梯度消失和梯度爆炸是什么？可以怎么解决?

   梯度消失主要是针对多层的神经网络而言的，对于后向传播求梯度，根据链式法则，下一层的梯度等于上一层的梯度乘上上一层的输入值对权重的求导。就会有多个梯度之间的连乘，当梯度是一个 0 到 1 之间的数的时候，每次后向传到第一层的时候，可能第一层的梯度已经近乎 0 了，导致了权重向量的变化值极小，这就是梯度消失。对于梯度爆炸就是梯度消失的一个反例，当梯度都很大的时候，连乘后第一层的梯度极大，导致了权重向量对输入的值很敏感，泛化能力弱，模型不合理，这就是梯度爆炸带来的不利影响。

References

[1] 归一化的影响 https://www.zhihu.com/question/20455227

[2] Stephen Boyd, Lieven Vandenberghe, 王书宁等译 *Convex Optimaization*, 清华大学出版社, 2013: 297-299

[3] 深度学习中的激活函数引导 https://www.cnblogs.com/ranjiewen/p/5918457.html

[4] 机器学习中的范数规则化 http://blog.csdn.net/zouxy09/article/details/24971995

[5] 数据归一化处理 http://blog.csdn.net/acdreamers/article/details/44664205

[6] 周志华，机器学习，清华大学出版社，2016:97-104