人工智能lab3实验报告

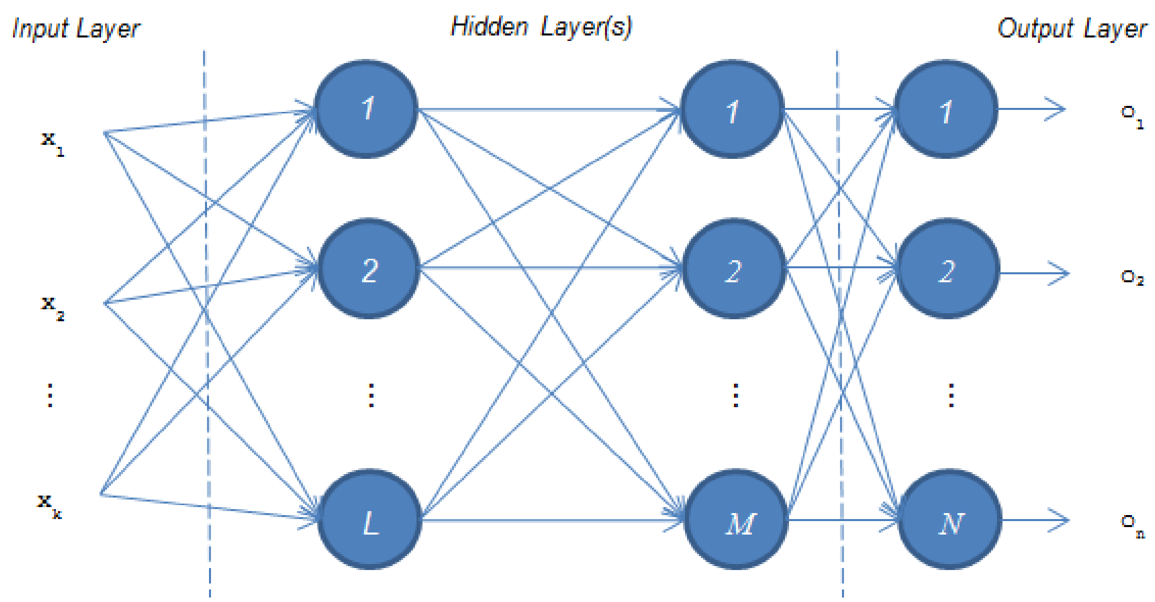
学号：17338233 专业：计科 姓名：郑戈涵

反向传播神经网络(BPNN)

算法原理

神经网络结构

反向传播神经网络是使用反向传播误差的方法训练的人工神经网络。人工神经网络由输入，激活函数，人工神经元构成，每个人工神经元能够接受一个输入，经过激活函数，然后输出结果。神经网络的结构如下，这次实验中使用一个隐藏层。



人工神经网络可以用于处理分类及回归问题。对于分类问题，输出层使用sigmoid函数，归一化后可以根据大小排序来分类；对于回归问题，输出层直接将隐藏层的α加权即可输出。

隐藏层的节点个数可以自己确定，可以使用下面的**经验公式**：

$$N_h = \frac{N_a}{\alpha * (N_i + N_o)}$$

其中：

N_i:输入层的神经元个数
N_o:输出层的神经元个数
N_i:训练集的样本个数
α:一个随机的比例系数，通常为2-10

训练步骤

使用训练集训练人工神经网络有以下步骤：

1. 取出训练集中的一个样本，将输入经过前向传播，得到输出结果
2. 将输出结果与与样本的结果计算得到误差Error
3. 将误差通过网络反向传播，计算出各个层的梯度

4. 使用梯度更新权值矩阵

计算误差使用最小二乘方法：

$$C = \frac{|\hat{y} - y|^2}{2}$$

y为预测结果，$\hat{y}$为实际结果。

则误差关于y的梯度为：

$$\frac{\partial C}{\partial y} = y - \hat{y}$$

由于该模型涉及到的参数较多，考虑到效率，一般使用矩阵处理参数，因此以下约定各个变量的字母：

z_k^l:第l层的第k个神经元的输入

z^l:第l层的输入向量

a_k^l:第l层的第k个神经元的输出

a^l:第l层的输出，即激活向量

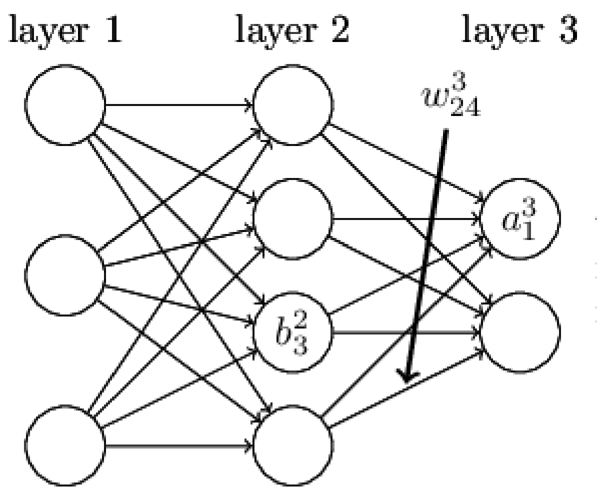
w_{jk}^l:第l层的第j个神经元到第$l-1$层的第k个神经元的权值

W^l:第$l-1$层到第l层的权值矩阵

b^l:第l层的偏置向量

σ:激活函数，本次实验使用sigmoid函数

上面约定中的层数都是将输入层作为第一层。



前向传播

前向传播有如下关系：

$$z^l = W^l \cdot a^{l-1} + b^l$$
$$a^l = \sigma(z^l)$$

反向传播

反向传播时有如下关系：

$$\frac{\partial C}{\partial w_{jk}^l}=\frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l}=\frac{\partial C}{\partial z_j^l}a_k^l$$

可以看出，$\frac{\partial C}{\partial z_j^l}$非常重要，因此令

$$\delta_j^l=\frac{\partial C}{\partial z_j^l}$$

由链式法则可知，输出层有如下关系（**L代表输出层**）：

$$\delta_j^L=\frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}$$

因为$a_j^L=\sigma(z_j^L)$，所以

$$\delta_j^L=\frac{\partial C}{\partial a_j^L}\sigma'(z_i^L)=\frac{\partial C}{\partial a_j^L}\sigma(z_i^L)(1-\sigma(z_i^L))$$

写成矩阵形式：

$$\delta^L=\nabla C_{a^L}\odot\sigma'(z^L) \tag{1}$$

若输出层没有进行激活，则$a_j^L=z_j^L$，那么

$$\delta_j^L=\frac{\partial C}{\partial a_j^L} \tag{2}$$

$\odot$代表**阿达玛乘积 (Hadamard product)**，即两个相同大小的矩阵对应元素乘积。

对于不同层的δ^l，可以证明如下结论

$$\delta^l=((W^{l+1})^T)\delta^{l+1})\odot\sigma'(z^l)$$

首先对于δ_j^l，由链式法则有

$$\begin{aligned}\delta_j^l=\frac{\partial C}{\partial z_j^l}&=\sum_k\frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l}\\&=\sum_k\frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial a_j^l}\sigma'(z_j^l)\\&=\sum_k\frac{\partial C}{\partial z_k^{l+1}}w_{kj}^{l+1}\sigma'(z_j^l)\\&=\sum_k\delta_k^{l+1}w_{kj}^{l+1}\sigma'(z_j^l)\end{aligned}$$

因此，上式写成矩阵形式：

$$\delta^l=\sigma'(z^l)\odot((W^{l+1})^T\delta^{l+1}) \tag{3}$$

对于C关于偏置b^l的偏导数，使用链式法则：

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \cdot 1$$

所以，写成矩阵形式可得：

$$\frac{\partial C}{\partial b^l} = \delta^l \tag{4}$$

对于C关于w_{ij}的偏导数，使用链式法则：

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$$

所以，写成矩阵形式可得：

$$\frac{\partial C}{\partial W^l} = \delta^l \cdot (a^{l-1})^T \tag{5}$$

综上，使用反向传播计算梯度共有四个式子(处理分类时使用(1)式，回归时使用(2)式)：

$$\delta^L = \nabla C_{a^L} \odot \sigma'(z^L) \tag{1}$$

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \tag{2}$$

$$\delta^l = \sigma'(z^l) \odot ((W^{l+1})^T \delta^{l+1}) \tag{3}$$

$$\frac{\partial C}{\partial b^l} = \delta^l \tag{4}$$

$$\frac{\partial C}{\partial W^l} = \delta^l \cdot (a^{l-1})^T \tag{5}$$

梯度下降

已知梯度则可以对权重矩阵进行更新了，更新公式为

$$W_{(i+1)}^l = W_{(i)}^l - \eta \frac{\partial C}{\partial W_{(i)}^l}$$
$$b_{(i+1)}^l = b_{(i)}^l - \eta \frac{\partial C}{\partial b_{(i)}^l}$$

每一层都是在计算出梯度后进行更新权值和偏置。上面公式中的各个量都可以通过反向传播的四个公式得出。

mini-batch

与逻辑回归类似，训练神经网络也有批梯度下降的概念，由于训练神经网络的速度更慢，一般来说使用的并不是整个训练集来训练网络，计算速度慢，收敛速度也不理想，一般使用SGD结合批梯度下降的方法，将整个训练集打乱顺序，并均分为数个batch，对于每个batch，计算神经网络在各个样本上的梯度，计算梯度的平均值来更新权值及偏置。遍历完所有的batch，则成为一个epoch，因此每经过一个epoch就相当于将完整的训练集都交给神经网络了。

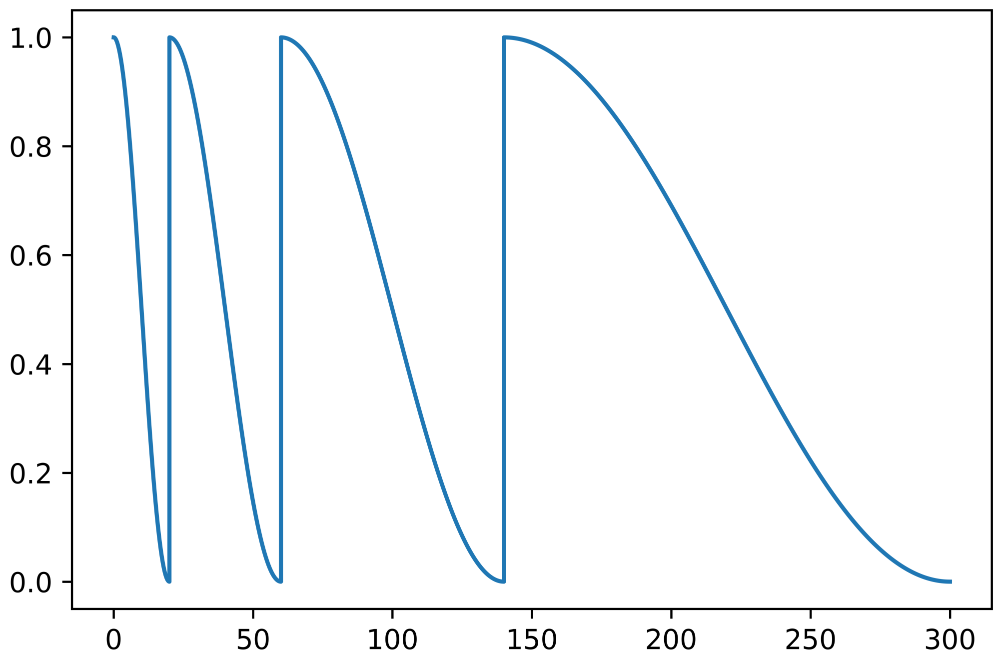
余弦退火(Cosine annealing)

学习率对训练神经网络的影响非常大，随着训练次数的增加，如果学习率保持不变，可能导致后期loss震荡剧烈，余弦退火是一个计算学习率的方法，第T_{cur}次批梯度下降时，若总epoch数为T_i，学习率可以使用下面的公式计算：

$$\eta_t = \eta^i_{min} + \frac{1}{2}(\eta^i_{max} - \eta^i_{min})(1 + \cos(\frac{T_{cur}}{T_i}\pi))$$

其中：$\eta^i_{max}, \eta^i_{min}$是学习率的范围。

学习率随训练次数的变化趋势如下：



该方法可以一定程度上避免收敛到局部最优解。

伪代码或者流程图

前向传播

注意因为是回归，最后一层不需要激活，返回的预测结果应该是激活前的。

```
procedure forward_feed():
input network: n
      training sample: sample
output: list of a and z(output)
    alist<-[]
    x<-sample
    foreach weight,b in n.weights
        z<-weight*x+b
        a<-sigmoid(z)
        append a into alist
        x<-a
    endfor
    alist[-1]<-z
return z,alist
```

反向传播

```
procedure backpropagation():
input: network: n
      training sample: sample
output: gradient of weights and biases
    prediction,alist<-forward_feed(n,sample)
    actual<-to(ts)
    error<-actual-prediction
    layer<-n.lastlayer
    delta[layer.no]=error
    layer<-layer's frontlayer
    while layer is not input layer
        delta[layer.no]<-sigmoidprime(alist[layer.no])*weights[layer.no+1]*delta[layer.no+1]
    endwhile
    foreach l in n.layer
        gradient_of_weight[l.no]<-delta[l.no]*alist[l.no]
        gradient_of_biase[l.no]<-delta[l.no]
    endfor
return gradient_of_weight,gradient_biase
```

mini-batch

```
procedure mini_batch():
input:network: n
      training set: ts
      actual output: to
output:network n
do
    shuffle(ts)
    divide ts into mini-batch
    batchsize<-len(mini-batch)
    foreach batch in mini-batch
        foreach sample in batch
            gradient<-gradient+(backpropagation(n,sample))
        gradient<-gradient/batchsize
        update weights and biases
    endfor
    calculate loss of training set
until loss is small enough
return n
```

代码展示

神经网络适合设计成类，包含前向传播，反向传播，SGD等各种方法，我设计了ANN类，主要部分如下：

```
class ANN:
    def __init__(self,sizes):
#          输入层之外的层数,例子为1+2，layNum=3
        self.num_layers =len(sizes)
        self.sizes=sizes
#          i索引w(i+1)矩阵
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
```

```
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]
    #前向传播，计算预测结果
    def forward_pass(self,x):
    #反向传播，计算对权值和偏置梯度
    def backPropagation(self,x,y):
    #随机批梯度下降，使用minibatch函数更新权值和偏置
    def SGD(self,train_set,batch_size,lr=1e-3,epochs=1e3):
    #普通的梯度下降
    def gradient_descent(self,x,y,lr=1e-3):
    #全训练集用于梯度下降
    def train(self,train_set,lr=1):
    #使用batch中的样本的梯度均值更新权值和偏置
    def minibatch(self,batch,lr=1e-3):
    #使用验证集计算方差
    def validate(self,validation_set):
```

ANN只需要保存偏置，权值矩阵，每层的节点个数和层数即可。该类实现的是**多层深度神经网络**，各个层的节点个数在构造神经网络时给出。

前向传播

前向传播中每一层都进行了激活，因为最后一层不需要，因此返回的是未激活的结果。

```
def forward_pass(self,x):
    a=np.matrix(x).transpose()
    for w,b in zip(self.weights,self.biases):
        z=w*a+b
        # 激活
        a=sigmoid(z)
    return float(z)
```

反向传播

反向传播函数返回对于一个样本的梯度。

```
def backPropagation(self,x,y):
    grad_w=[np.zeros(w.shape)for w in self.weights]
    grad_b=[np.zeros(b.shape)for b in self.biases]
    delta=[np.zeros(b.shape)for b in self.biases]
    a=np.matrix(x).transpose()
    alist=[a]
    zlist=[]
    #前向传播记录每一层的输出
    for w,b in zip(self.weights,self.biases):
        z=w*a+b
        a=sigmoid(z)
        zlist.append(z)
        alist.append(a)
    #修改最后一层的结果，因为处理的是回归问题
    alist[-1]=zlist[-1]
    #算法原理的公式(1)
    delta[-1]=(z-y).transpose()
    #从最后一次开始倒着更新delta，delta对应算法原理中的$delta$
    for l in range(2,self.num_layers):
        # 算法原理的公式(2)
```

```
        delta[-1]=np.multiply(sigmoid_prime(alist[-1]),self.weights[1-1].transpose()*delta[1-1])
        for i in range(self.num_layers-1):
            # 算法原理的公式(4)
            grad_w[i]=delta[i]*np.mat(alist[i]).transpose()
            # 根据算法原理的公式(3)，偏置的梯度不需要计算，可以直接返回delta
            return grad_w, delta
```

mini-batch

这部分涉及两个函数，**SGD和minibatch**。

SGD函数创建batch，并调用minibatch函数进行权值和偏置的更新。

```
def SGD(self,train_set,validation_set,batch_size,lr=1e-3,epochs=1e3):
    # 初始化最优的方差和神经网络
    variate=float('inf')
    besta=None
    # 进行epoch次遍历，每次将整个训练集传给神经网络
    for j in range(int(epochs)):
        # 打乱训练集的顺序
        np.random.shuffle(train_set)
        # 步长为batch_size，将训练集分成（样本个数/batch_size）个batch，存入mini_batches中
        mini_batches = [train_set[k:k+batch_size] for k in range(0, train_set.shape[0], batch_size)]
        # 对每个mini_batch,训练网络
        for mini_batch in mini_batches:
            self.minibatch(mini_batch, eta)
        # 计算验证集上的方差
        curVar=self.validate(validation_set)
        # 记录最好的方差和网络
        if curVar<variate:
            variate=curVar
            besta=copy.deepcopy(self)
        # 进行一轮更新后，输出网络的属性
        print("Epoch {} : {}, best : {}".format(j,curVar,variate));
    return besta
```

梯度下降

gradient_descent函数使用单个样本计算出梯度函数后，直接使用学习率更新权值和偏置。train函数遍历训练集的每个样本进行训练。

```
def gradient_descent(self,x,y,lr=1e-3):
    # 计算梯度
    grad_w,grad_b=self.backPropagation(x,y)
    # 更新权值和偏置
    self.weights=[w-lr*gw for w,gw in zip(self.weights,grad_w)]
    self.biases=[b-lr*gb for b,gb in zip(self.biases,grad_b)]
def train(self,train_set,lr=1):
    # 整个训练集进行梯度下降
    for row in train_set:
        self.gradient_descent(row[:-1],row[-1],lr)
```

验证函数

validate函数将输入通过前向传播获得的结果与真实结果计算方差并返回。

```
def validate(self,validation_set):
    arr=[self.forward_pass(row[:-1])for row in validation_set]
    return np.var((np.matrix(arr)-validation_set[:,-1]))
```

数据预处理

由于数据集中有日期这样的数据，并且年份的数字较大，为了方便训练，需要进行预处理。

```
def preprocess(dataSet):
    dataSet.dteday=dataSet.dteday.map(date2int)
    # 计算最大的date用于归一化
    maxdte=float(max(dataSet.dteday))
    dataSet.dteday=dataSet.dteday.map(lambda x:float(x)/maxdte)
    return dataSet.values[:,1:]

def date2int(str):
    # 取出被'/'分隔的各个元素
    l=[int(x) for x in str.split('/')]
    return l[0]+l[1]*30+l[2]
```

date2int函数将日期字符串处理为数字，preprocess函数将数据集中的日期转换成对应的数字。

划分数据集

该函数按照比例将数据集随机划分为训练集和验证集并返回。

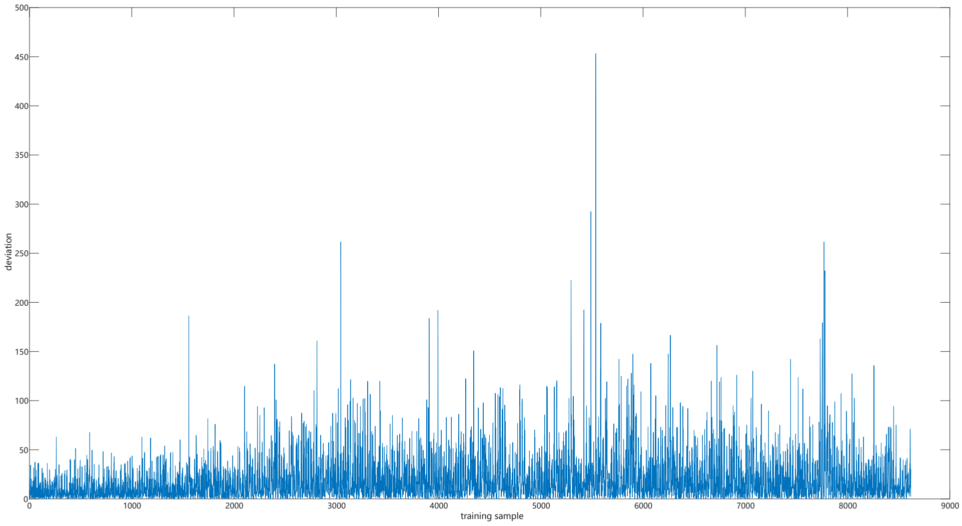
```
def split(dataSet,proportion):
    # 对数据集进行洗牌
    np.random.shuffle(dataSet)
    train_num=int(dataSet.shape[0]*proportion)
    return dataSet[0:train_num,:],dataSet[train_num:,:]
```

本次实验中我将数据集的80%作为训练集，剩下的作为验证集，选出验证集上效果最好的模型。

实验结果以及分析

结果展示和分析

下面是训练集上训练10000个epoch后在训练集上的结果，预测均方差为900左右，即平均有30左右的误差。

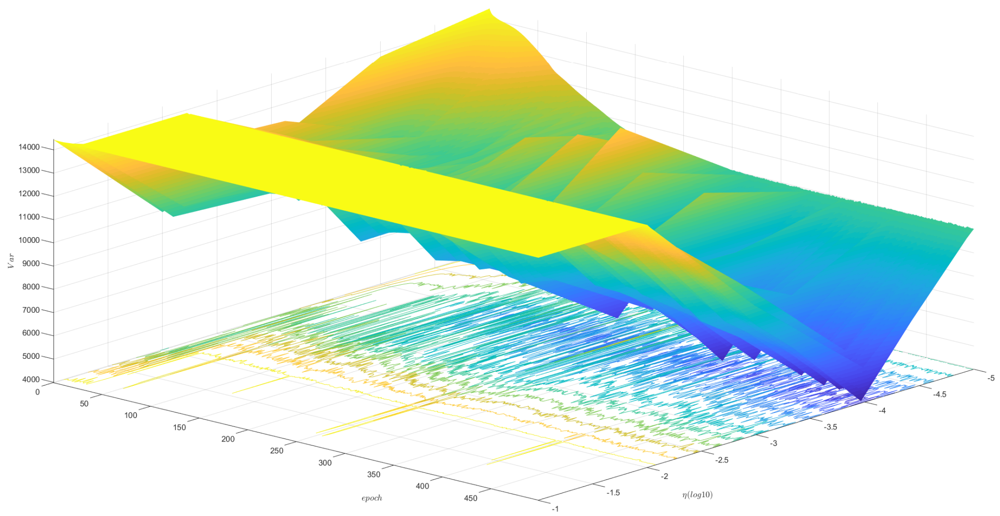


可以看到，除了部分特殊明显偏差较大的样本的预测值较差外，大部分的误差都是比较小的，说明模型没有错误。

模型性能展示和分析

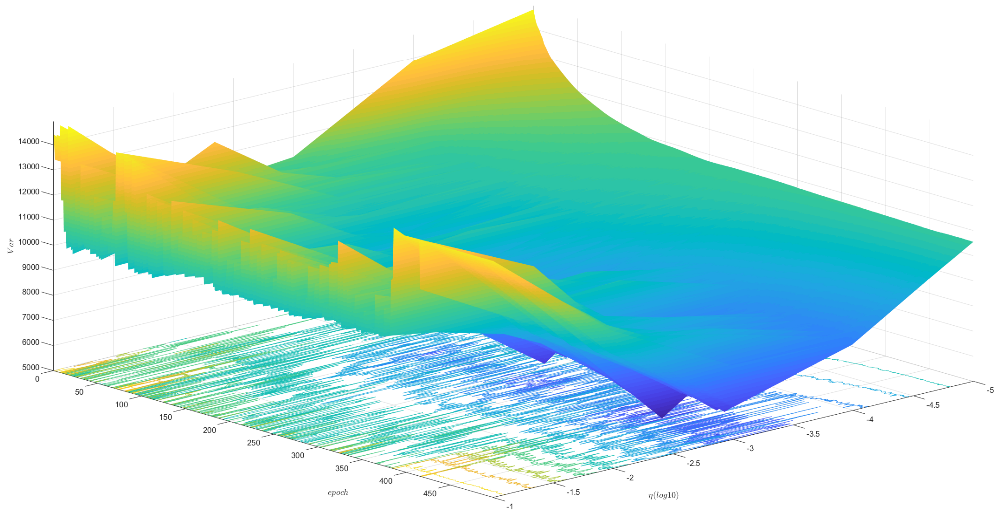
学习率

学习率对训练速度及loss都会有显著影响，下图是不同学习率下loss随epoch数变化的情况，使用两个隐藏层，各有80个节点，可以看到，当学习率为0.1时，loss几乎没有变化，出现了**梯度消失**的问题，而当学习率接近10^{-4}，loss下降非常明显，学习率接近10^{-5}，loss下降比较缓慢。因此，训练开始时选择接近10^{-4}的学习率比较合适。



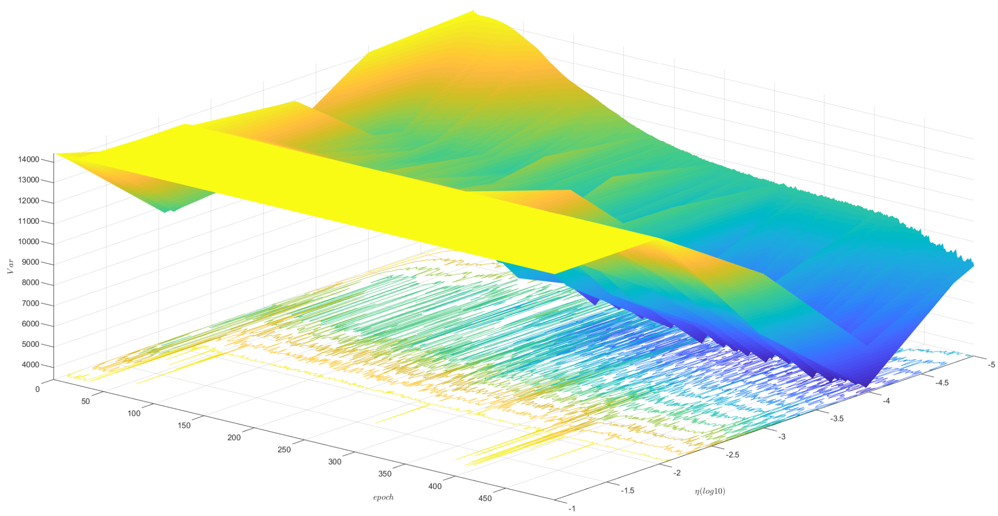
隐藏层个数

尝试隐藏层个数不同的神经网络（单层，三层），每层均为80个节点，明显可以感觉层数多的神经网络训练更慢，因为计算的矩阵规模增加，单个隐藏层的结果如下：



可以看到，对于单个隐藏层，较大的学习率会导致较大的震荡，学习率为10^{-3}的效果比较好。总体来看，loss的下降速度不如两个隐藏层。还可以注意到，单个隐藏层学习率为10^{-1}时不会出现**梯度消失**问题。

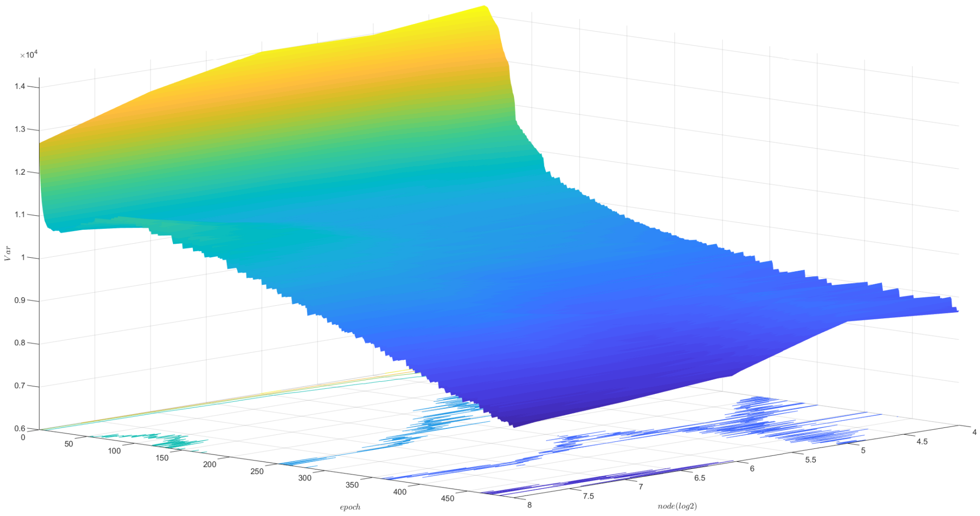
三个隐藏层的结果如下：



可以看到，和两个隐藏层的结果相似，学习率较大时几乎不动，因为对于sigmoid函数，梯度会因为计算导数的链式法则而相乘，层数增加更容易导致出现梯度消失。学习率接近10^{-4}最好，此时loss的下降速度比两个隐藏层的稍好，但是没有明显差距。总体来说，三层的神经网络loss有更快的下降速度，缺点则是计算规模大，训练速度慢。

单隐藏层节点个数

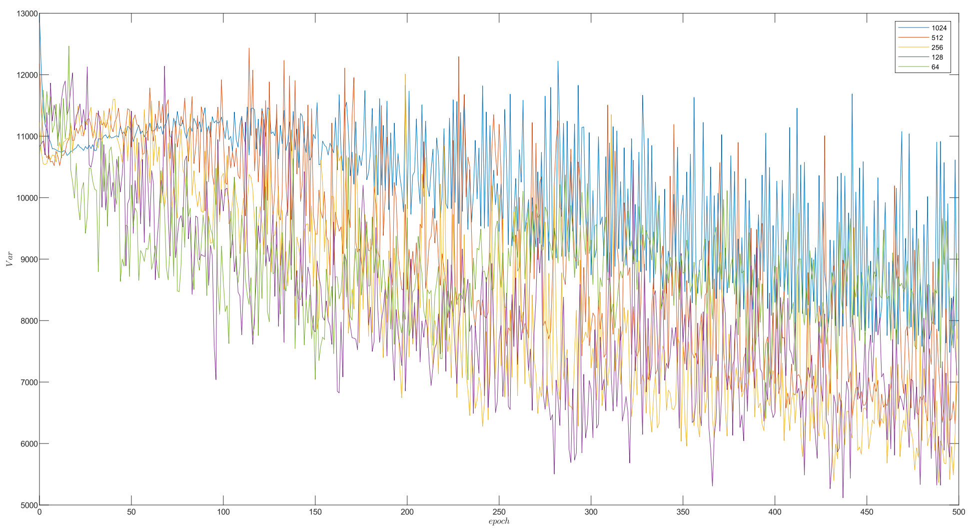
尝试单个隐藏层不同节点个数的神经网络训练500个epoch，训练速度也有一定的差别，loss如下：



因为使用的是上面测试得到的较好的学习率（10^-4），所以loss都有明显的下降。其中可以注意到，节点数较多的loss相比较少的神经网络在一开始就有大幅度的下降，而之后loss下降趋势大致相同，因此总体上是节点数较多的神经网络loss下降最多。但是从图中也可以看出，当节点超过64后，增加节点的收益开始下降。考虑到训练速度，可以选择64个节点。

mini-batch

不同大小的mini-batch也会对训练速度产生影响。下图是对于不同batch size使用较好的学习率（10^{-4}）训练500个epoch的loss图：



可以看到，1024的效果是很不好的，而batch size比较小时，训练后期下降幅度不明显，本次试验中使用128的效果比较好。

使用最优的参数训练得到的神经网络loss为900，训练结果在**结果展示与分析**小节进行了展示。

思考题

尝试说明下其他激活函数的优缺点

一般神经网络中用的激活函数有三种，sigmoid，tanh，ReLU。

sigmoid

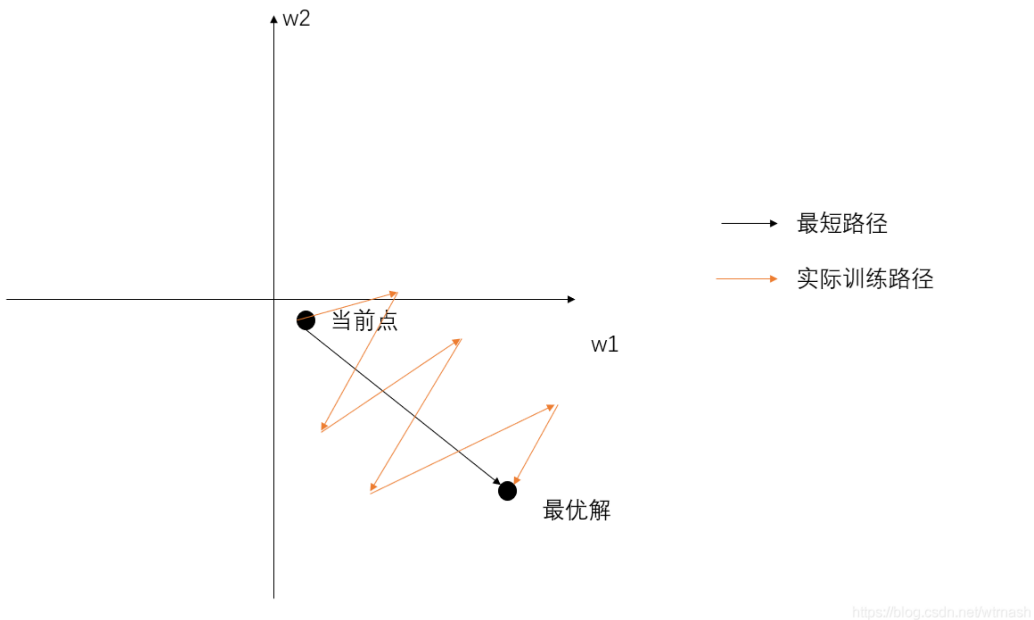
$$\sigma(x)=\frac{1}{1+e^{-x}}$$

优点

1. 平滑，便于求导 。
2. 输出范围有限，在分类时可以作为输出层。

缺点

1. 计算幂，计算复杂度高。
2. 对输入的微小变化不敏感，容易出现梯度消失的问题。
3. 不是0均值，会出现z型更新的现象，如下图。



tanh

$$\tanh x=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

优点

1. 平滑，便于求导 。
2. 输出范围有限，在分类时可以作为输出层。
3. 是0均值的。
4. 数据幅度会被压缩，不会随层数无限扩大。

缺点

1. 计算幂，计算复杂度高。
2. 容易出现梯度消失的问题。

ReLU

$$f(x) = \max(0, x)$$

优点

1. 收敛速度快。
2. 不会出现梯度饱和或消失的问题。
3. 计算复杂度低。

缺点

1. 不是0均值的。
2. x<0时，梯度为0的神经元及之后的神经元的梯度永远为0，对应的权值等参数不会被更新。学习率高的时候容易出现该问题。
3. 数据幅度不会被压缩，会随层数无限扩大。

有什么方法可以实现传递过程中不激活所有节点?

1. 一个节点的作用对应于权值矩阵的一行，只要每次使用梯度更新权值矩阵时，将不激活的节点对应的行设置为0即可。
2. 使用饱和区比较平缓的函数（比如sigmoid）作为激活函数，当输出接近饱和区时，梯度几乎为0，节点会处于不激活的状态。

梯度消失和梯度爆炸是什么？可以怎么解决?

梯度消失

当激活函数接近平缓区域（饱和区）时，导数接近0，根据链式法则，当前层导数等于后面各层导数的乘积，因此数量级进一步减小，导致梯度几乎为0。

梯度爆炸

和梯度消失类似，如果各层导数较大，那么前面的层的梯度会非常大，权值矩阵会有大幅度变化，可能导致溢出。

解决方法

两种方法本质上是因为计算复合函数导数符合链式法则，层数较多和激活函数饱和时数值过小导致的。可以用以下方法解决：

1. 如果可以，使用较少的隐藏层数
2. 使用ReLU等不会出现该问题的激活函数
3. 设置阈值，限制梯度的大幅度增加
4. 使用包含权重的正则化项进行惩罚