四次课后作业题目：
作业 1
习题 2-2 画出野人与传教士问题的状态空间图。

这一题主要是让学生熟悉状态空间法的使用，培养用人工智能基础方法来解决问题的思想和能力

我们规定每层节点的运输优先规则（移动先后顺序）：1.两个传教士 2.两个野人 3.一个野人一个传教士 4.一个传教士 5.一个野人

对问题作如下抽象，以列表 state=[a,b,c]分别代表初始岸边的传教士人数，野人人数，船只数目，有

初始状态：state=[3,3,1]

agent 所有可能行动：

当 c==1 时,在以下五种状态中选择一种执行：（运输执行顺序：1.两个传教士 2.两个野人 3.一个野人一个传教士 4.一个传教士 5.一个野人）

注意，把 c=1 变成 c=0

a=a-2,c=0

b=b-2,c=0

b=b-1,a=a-1,c=0

a=a-1,c=0

b=b-1,c=0

但是需保证执行动作 state 处于状态空间之中，否则不能执行。

当 c==0 时,在以下五种状态中选择一种执行：（执行顺序：1.两个传教士 2.两个野人 3.一个野人一个传教士 4.一个传教士 5.一个野人）

注意，把 c=0 变成 c=1

a=a+2,c=1

b=b+2,c=1

b=b+1,a=a+1,c=1

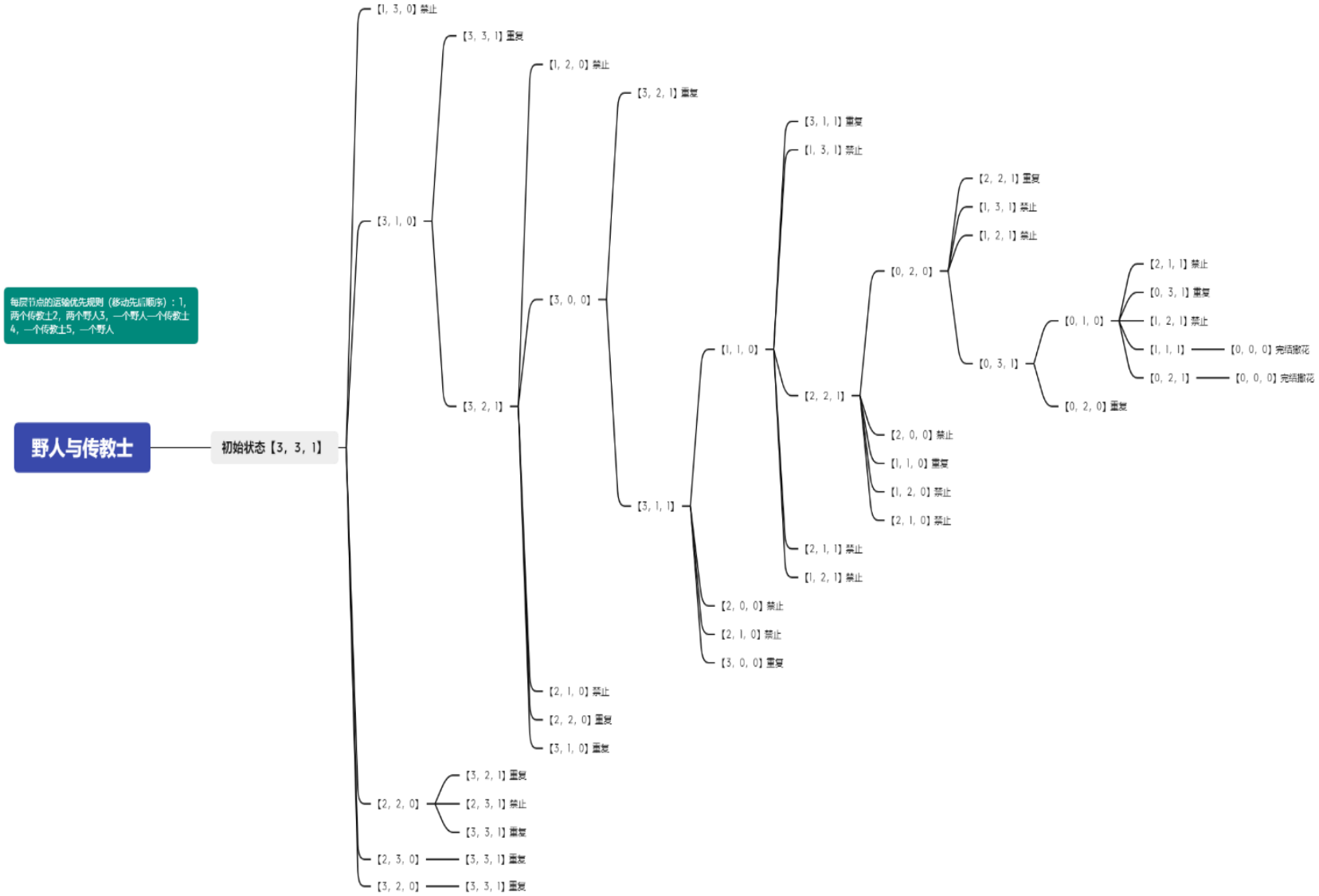
a=a+1,c=1

b=b+1,c=1

但是需保证执行动作 state 处于状态空间之中，否则不能执行。

状态空间：

[3,3,1],[3,2,1],[3,1,1],[3,0,1],[2,2,1],[1,1,1],[0,3,1],[0,2,1],[0,1,0]
[0,0,0],[3,2,0],[3,1,0],[3,0,0],[2,2,0],[1,1,0],[0,3,0],[0,2,0],[0,1,0]


每层节点的运输优先规则（移动先后顺序）：1，两个传教士2，两个野人3，一个野人一个传教士4，一个传教士5，一个野人
野人与传教士
初始状态【3，3，1】
【1，3，0】禁止
【3，1，0】
【3，3，1】重复
【3，2，1】
【1，2，0】禁止
【3，0，0】
【3，2，1】重复
【3，1，1】
【1，1，0】
【3，1，1】重复
【1，3，1】禁止
【2，2，1】
【0，2，0】
【2，2，1】重复
【1，3，1】禁止
【1，2，1】禁止
【0，3，1】
【0，1，0】
【2，1，1】禁止
【0，3，1】重复
【1，2，1】禁止
【1，1，1】
【0，0，0】完结撒花
【0，2，1】
【0，0，0】完结撒花
【0，2，0】重复
【2，0，0】禁止
【1，1，0】重复
【1，2，0】禁止
【2，1，0】禁止
【2，1，1】禁止
【1，2，1】禁止
【2，0，0】禁止
【2，1，0】禁止
【3，0，0】重复
【2，1，0】禁止
【2，2，0】重复
【3，1，0】重复
【2，2，0】
【3，2，1】重复
【2，3，1】禁止
【3，3，1】重复
【2，3，0】
【3，3，1】重复
【3，2，0】
【3，3，1】重复

作业 2
习题 5-8 试实现一个分层前馈神经网络的数据结构

这一题主要是增强学生对于神经网络前向计算过程的理解，有助于进一步学习复杂神经网络的结构和原理

假定网络层为三层，输入大小为 2，隐层神经元为 2

```
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_auc_score
from sklearn.metrics import mean_squared_error
import matplotlib
matplotlib.use("TkAgg")
np.random.seed(11)

class FFNN(object):#网络分为 3 层，输入大小为 2，隐藏层神经元为 2
def __init__(self,input_size=2,hidden_size=2,output_size=1):
    self.input_size = input_size +1
    self.hidden_size= hidden_size +1
    self.output_size = output_size

    self.o_error = 0
    self.o_delta = 0
    self.z1 =0
    self.z2 =0
    self.z3 =0
    self.z2_error = 0
    #输入层到隐层的权重矩阵
    self.w1 = np.random.randn(self.input_size,self.hidden_size)
    #隐层到输出层的权重矩阵
    self.w2 = np.random.randn(self.hidden_size,self.output_size)



def sigmoid(s):
    return 1/ (1+np.exp(-s))#激活函数 sigmoid

def sigmoid_prime(s):
    return sigmoid(s) * (1-sigmoid(s))#sigmoid 函数的导数

```

```
#向前传播函数：使用输入和权重之间的点积来计算输出，然后将所有内容通过 sigmoid 传递
def forward (self, X):
    X['bias']=1 #向输入添加 1 作为权重的偏差
    self.z1 = np.dot(X, self.w1) # X（输入） 与第一层权重的点积
    self.z2=sigmoid(self.z1) #激活函数
    self.z3=np.dot(self.z2,self.w2)#隐层(z2) 与第二层权重的点积
    o = sigmoid(self.z3) #最终激活函数
    return o



#用一个函数计算向前传递

def predict (self,X):
    return forward(self,X)#前向传播是我们将用于预测的内容，常为它创建别名 predict

#反向传播函数：误差的反向传播，调整权重并减小误差
#从输出开始，计算预测值与实际输出之间的误差，这将用于计算在更新权重时使用的 delta
#在所有层中，我们将神经元的输出用作输入，将其通过 sigmoid 的导数，然后乘以误差和步长（也称为学习率）
def backward (self,X,y,output,step):
    X['bias']=1 #向输入添加 1 作为权重的偏差
    self.o_error = y -output #计算一次误差
    self.o_delta = self.o_error * sigmoid_prime(output) * step
    # 对误差应用 sigmoid 导数
    self.z2_error =self.o_delta.dot(self.w2.T)#z2 误差:隐层权重对输出误差的影响
    self.z2_delta = self.z2_error * sigmoid_prime(self.z2) * step
    # 对 z2 误差使用 sigmoid 函数
    self.w1 += X.T.dot(self.z2_delta) #调整第一层权重
    self.w2 += self.z2.T.dot(self.o_delta)#调整第二层权重

    #就每个数据点训练模型时，进行两次传递，向前一次和反向一次
    def fit(self,X,y,epochs =10, step=0.05):
        for epoch in range(epochs):
            X['bias'] = 1 #向输入添加 1 作为权重的偏差
            output = self.forward(X)
            self.backward(X,y,output,step)
```

以上多层前馈神经网络模型，可较好的应用于一个 XOR（逻辑异或）的二元分类任务。
准备训练集和测试集：

```
#分离训练集和测试集中的数据集
msk=np.random.rand(len(data))<0.8
#大约 80%的数据将进入训练集
train_x,train_y = data[['x1','x2']][msk], data[['type']][msk].values
```

```
#其余的数据进入测试集
test_x,test_y = data[['x1', 'x2']][~msk],data[['type']][~msk].values

#x 训练网络如下：
my_network = FFNN()
my_network.fit(train_x,train_y,epochs-10000,step=0.001)

#验证算法性能如下：
pred_y = test_x.apply(my_network.forward,axis=1)
#重塑数据
test_y_ = [i[0] for i in test_y]
pred_y_ = [i[0] for i in pred_y]

print('MSE: ', mean_squared_error(test_y_,pred_y_))  #MSE 作为损失函数
print('AUC: ',roc_auc_score(test_y_,pred_y))
```

AUC 是一个模型评价指标，只能用于二分类模型的评价，对于二分类模型，还有损失函数（logloss），正确率（accuracy），准确率（precision），但相比之下 AUC 和 logloss 要比 accuracy 和 precision 用的多，原因是因为很多的机器学习模型计算结果都是概率的形式，那么对于概率而言，我们就需要去设定一个阈值来判定分类，那么这个阈值的设定就会对我们的正确率和准确率造成一定成都的影响。或者用混淆矩阵判定预测的好坏,XOR 问题结果去别明显，设阈值为 0.5 即可：

```
threshould = 0.5
pred_y_binary = [0 if i > threshould else 1 for i in pred_y_]
cm = confusion_matrix(test_y_,pred_y_binary,labels=[0.1])
print(pd.DataFrame(cm,index=['True 0','True 1'], columns=['Predicted 0', 'Predicted 1']))
```

使用 Keras 版：

使用 Keras 包实现 FFNN（要用到 Sequential 模型，Sequential 函数具体介绍参考 https://www.cnblogs.com/wj-1314/p/9579490.html）

初始化 Sequential 后需要添加的主要函数有：一，密集层 Dense（），目的是让所有神经元与下一层神经元都有一个连接，其输入参数为：1.指定神经元数量 2.输入纬度 input_dim（几个变量就是几），3.激活函数 activation，4.初始权重值 kernel_initializer）；二，add(),用于将层添加到模型；三，complie（）,编译模型，参数有：1.损失函数 2.梯度下降策略，即优化器，并指定学习率（可以不指定）；四，fit 方法来训练网络，参数有：1.训练数据 2.训练数据的标签 3.epoch 数（迭代次数，一个 epoch 包括整个数据集前向和反向通过网络）4.批量大小（batch size）,即训练集中用于一次梯度迭代的部分；五，数据洗牌（shuffle）,一般 shuffle=True,因为较大的 batch size 会导致局部最小值问题，失去在训练集外的泛化能力，如果 shuffle=True 可以使每次迭代更改批次，避免局部最小值。

对于二分类预测可以使用 Keras 包，需要一个输入神经元，三个隐层神经元和一个输出神经元。

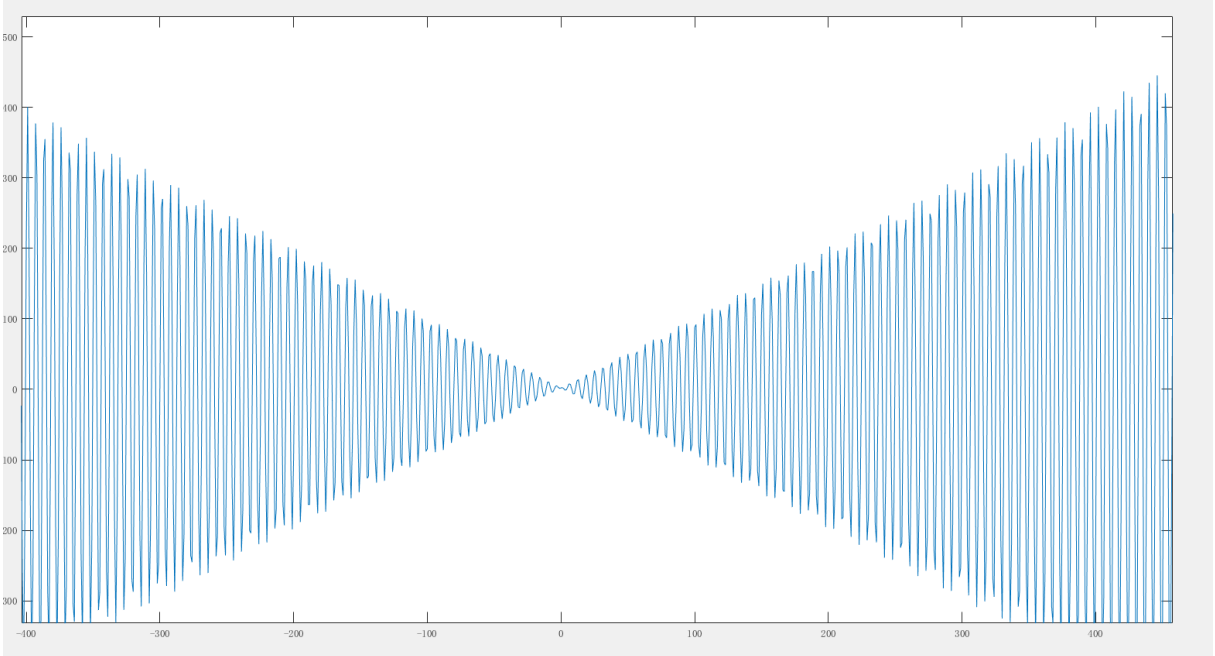
如果不是二分类问题需要重新规定合适的输入层隐藏层输出层的神经元个数，这里仅以二分类为例。

```
#Keras

from keras.models import Sequential
from keras.layers.core import Dense,Dropout,Activation
```

```
from keras.optimizers import SGD
from sklearn.metrics import mean_squared_error
import os
from keras.callbacks import ModelCheckpoint,Callback,EarlyStopping,TensorBoard

model = Sequential()
model.add(Dense(2, input_dim=2))#指定隐藏层的输入，XOR 情况下输入为 2，隐层神经元数目为 2

model.add(Activation('tanh'))#激活函数选用 tanh

#添加具有一个神经元的另一个全连接层，该层激活函数为 sigmoid
model.add(Dense(1))
model.add(Activation('sigmoid'))

#使用 SGD 作为优化方法来训练神经网络
sgd = SGD(lr = 0.1)
#编译神经网络，以 MSE 作为损失函数
model.compile(loss='mse', optimizer=sgd)
#训练网络，不在乎批次大小，运行两个 epoch
model.fit(train_x[['x1','x2']],train_y,batch_size=1, epochs=2)

#在测试集对 MSE 进行测量
pred = model.predict_proba(test_x)
print('NSE:',mean_squared_error(test_y,pred))
```

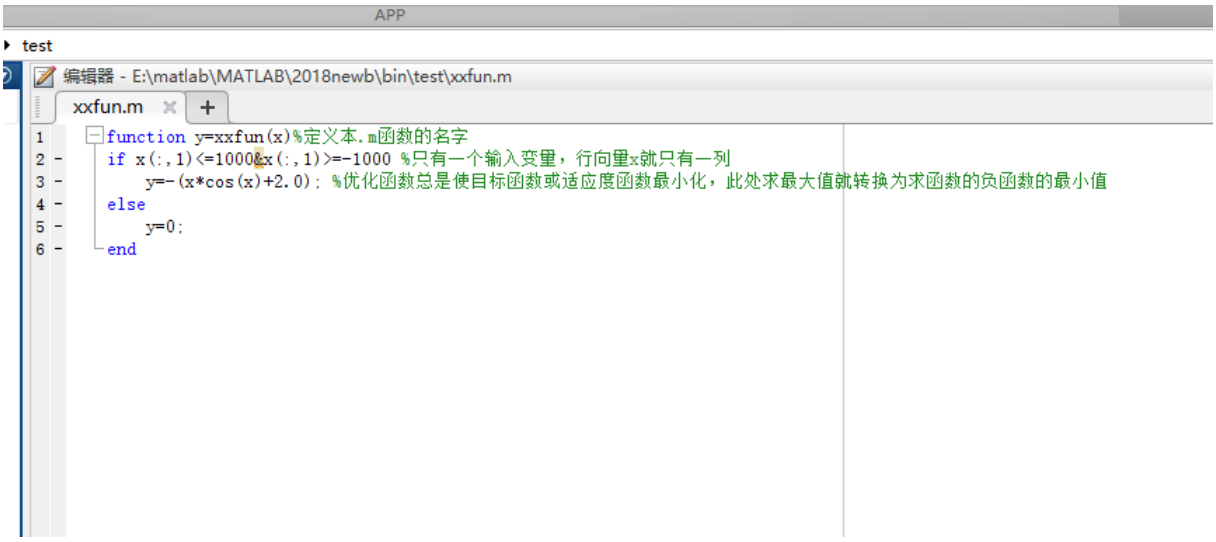
作业 3

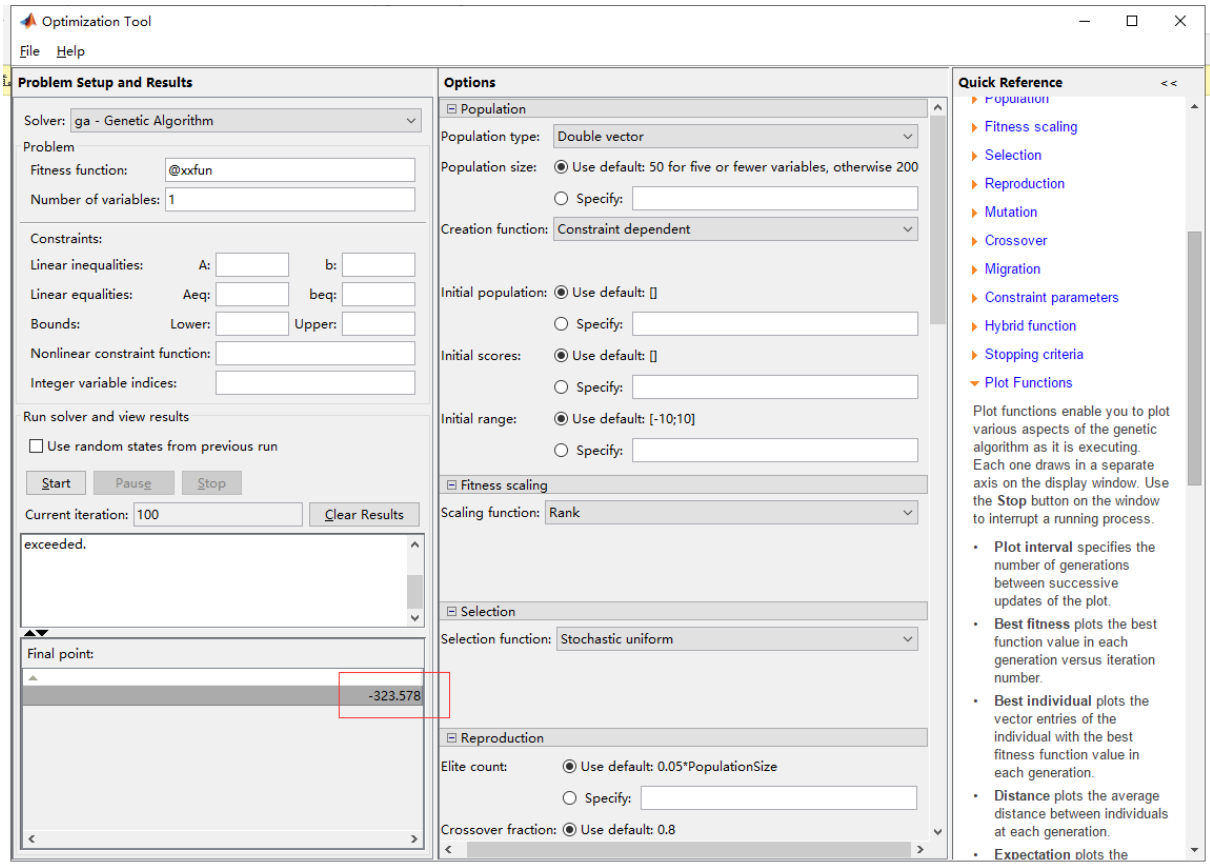
习题 5-16 用遗传算法求 F(x)=xcosx+2 的最大值



先画个图看看，发现这个最大值确实没有具体的数值，只能说在一定范围内有，那我们设置 x 范围在[-1000,1000]

用 matlab 遗传算法的 GUI 工具箱试试



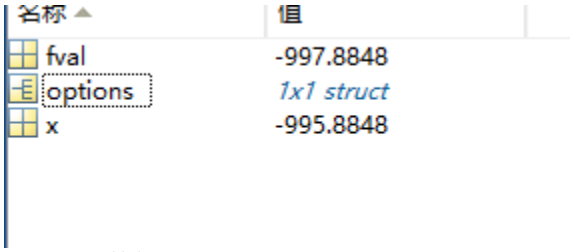


Use random statesf rom previous run 不选是为了充分利用遗传算法随机搜索的优点

```
>> [x,fval]=ga(@xxfun,1)
Optimization terminated: maximum number of generations exceeded.

x =

 -304.7457


fval =

 -306.7266
```

在默认参数下显然答案不怎么准确，可以改变代数、交叉概率来接近正确值，设定 200 代，0.3 交叉概率

```
>> options=gaoptimset('Generation',200,'CrossoverFraction',0.3);
>>  [x,fval]=ga(@xxfun,1,options);
Optimization terminated: average change in the fitness value less than options.FunctionTolerance.
>>
```



Fval：函数极小值　　x：x 的值
此处可以说 F(x)=xcosx+2 最大值为 997.8848

这一题是对上机题遗传算法解 TSP 问题的补充，通过求函数极值的简单问题锻炼智能算法编程能力，理解遗传算法求解不同问题的思路和问题表示方法。

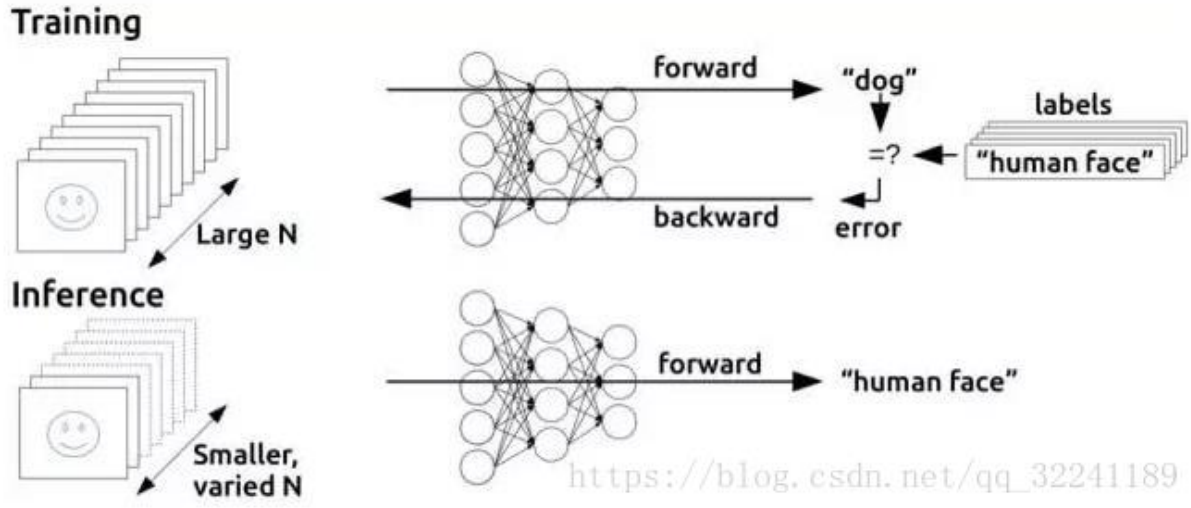
作业 4
习题 7-9 说明 BP 学习算法的原理，有能力的同学写出程序
这一题是对神经网络反向传播过程的巩固，要求学生理解梯度下降的计算过程以及神经网络参数的训练过程，深入理解神经网络的原理。

神经网络主要是由三个部分组成的,分别是:1) 网络架构 2) 激活函数 3) 找出最优权重值的参数学习算法。

BP 算法就是目前使用较为广泛的一种参数学习算法。

BP(back propagation)神经网络是 1986 年由 Rumelhart 和 McClelland 为首的科学家提出的概念，是一种按照误差逆向传播算法训练的多层前馈神经网络。

既然我们无法直接得到隐层的权值，能否先通过输出层得到输出结果和期望输出的误差来间接调整隐层的权值呢？BP 算法就是采用这样的思想设计出来的算法，它的基本思想：学习过程由信号的正向传播(求损失)与误差的反向传播(误差回传)两个过程组成。如图 1 所示为 BP 算法模型示意图。



BP 算法的一般流程

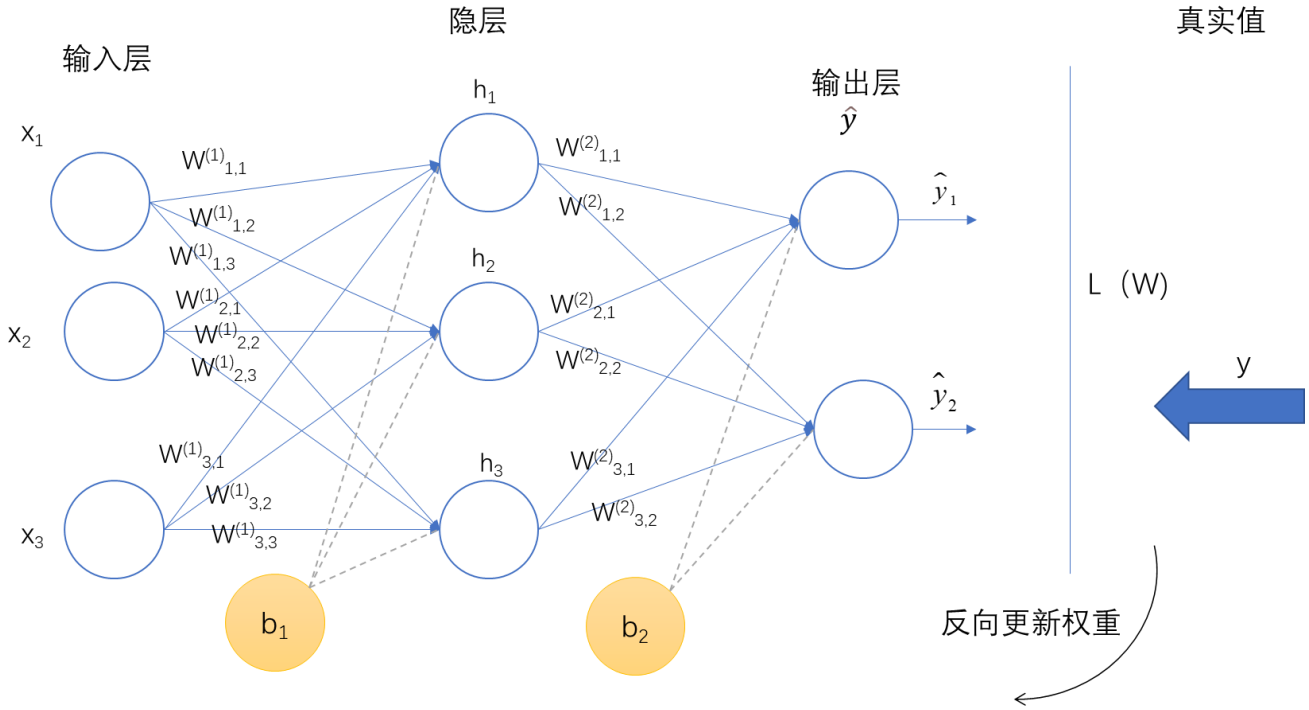
神经网络的训练是通过调整隐层和输出层参数，使得神经网络计算出来的结果 y 与真实结果 y 尽量接近，神经网络的训练主要包括正向传播和反向传播两个过程。

根据 BP 算法的基本思想,可以得到 BP 算法的一般过程：

1) 正向传播 FP(计算输出，为求损失做铺垫)。在这个过程中,我们根据输入的样本,给定的初始化

权重值 W 和偏置项的值 b,计算最终输出值以及输出值与实际值之间的损失值。如果损失值不在给定的范围内则进行反向传播的过程;否则停止 W,b 的更新。

以两层神经网络为例（一组输入层，一组隐层，一组输出层），计算方法通常为：输入层到隐层的权重矩阵 $W^{(1)}$的转置乘以输入向量 x，加上输入层到隐层的偏置 b1,将该值经过激活函数（一般是 sigmoid 函数），得到隐层的输出 h；同样的方法算输出层在激活函数之前的输出，即隐层到输出层的权重矩阵 $W^{(2)}$ 的转置乘以输入向量 h 再加上隐层到输出层的偏置 b2,并把它经过激活函数（这个“它”是一个中间向量，暂且用 z^l 表示，我们把 z^l 称作第 l 层的带权输入），可以得到最终输出。



（PS：多层前馈神经网络则隐层可以超过一层，相应的，随着层数增加神经网络的参数也显著增多：相邻两层之间必有一个偏置参数，权重参数的个数等于这两层的神经元个数之积。）

2) 反向传播 BP(回传误差)。将输出以某种形式通过隐层向输入层逐层反传,并将误差分摊给各层的所有单元，从而获得各层单元的误差信号,此误差信号即作为修正各单元权值的依据。**简而言之，就是首先根据神经网络正向传播计算出的输出值和期望值计算损失函数的值，然后计算损失函数对每个权重或偏置的偏导（计算梯度得到每个参数应调整的偏移量），最后进行参数的更新。**

由于 BP 算法是通过传递误差值 δ 进行更新求解权重值 W 和偏置项的值 b,所以 BP 算法也常常被叫做 δ 算法。

反向传播的核心是对损失函数 C 关于任何权重 w（或者偏置 b）的偏导函数 $\frac{\partial C}{\partial w}$ 的表达式。该表达式用于计算改变权重和偏置时损失变化的快慢。

常见使用均方误差（简称 MSE，又叫做二次代价函数，代价函数等等）作为损失函数：

$$C(w,b)=\frac{1}{2n}\sum_x\left\|y(x)-a\right\|^2$$

w 是神经网络中所有权重的集合，b 是所有偏置，n 是训练输入数量，a 表示输入为 x 时输出的向量（输出值），y(x)是目标输出（期望值）。如果学习算法能找到合适的权重和偏置，使得

$C(w,b)\approx 0$ 就能很好的工作；反之如果 $C(w,b)$ 很大则输出值和期望值差距很大，效果就不好，我们要最小化 $C(w,b)$（即找 C 的全局最小值），这可以通过梯度下降法实现。因为函数 C 通常是一个复杂的多元函数，无法用微积分（开导）来找到最小值，因而引入梯度下降算法。

梯度下降算法和随机梯度下降的计算过程：(理解梯度下降的计算过程）

假设要最小化某函数 $C(v)$，它可以是任意的多元实值函数，其中

$$\Delta C=\frac{\partial C}{\partial v_1}\Delta v_1+\frac{\partial C}{\partial v_2}\Delta v_2 \quad \text{.....（1）}$$

。用 v 代替 w和b 是强调这可以是任意函数，不局限于神经网络。为最小化 $C(v)$，假设 C 是一个只有两个变量 v_1,v_2 的函数。想象一个小球从山谷斜坡滚落，经验告诉我们小球一定会滚到谷底，我们试图利用这一想法找到函数最小值。小球随机选择一个起始位置，模拟小球滚落谷底的运动，可以通过计算 C 的导数（或者二阶导数）来简单模拟，这些导数能描述山谷局部形状，依此获得小球的滚动情况。

思考：如果我们能自己设置物理定律支配小球的滚动方式，我们会运用什么运动规律来让小球总能够滚落到谷底呢？

当在 v_1和v_2 方向分别将小球移动一个很小的量，即 Δv_1 和 Δv_2 时，利用微积分可知 C 将如下变化：

$\Delta v=(v_1,...v_m)^T$ 。我们要寻找一种选择 Δv_1 和 Δv_2 的方法使得 ΔC 为负，即选择它们使小球滚落谷底。

对于如何选择，需要定义 Δv 为 v 变化的向量，$\Delta v\equiv(\Delta v_1,\Delta v_2)^T$ .....（2），其中 T 是转置符号，用于将行向量转化为列向量。定义 **C 的梯度**为偏导数的向量，$\left(\frac{\partial C}{\partial v_1}\Delta v_1,\frac{\partial C}{\partial v_2}\Delta v_2\right)^T$。用 ∇C 表示梯度向量：$\nabla C\equiv\left(\frac{\partial C}{\partial v_1}\Delta v_1,...,\frac{\partial C}{\partial v_m}\Delta v_m\right)^T$ .....（3）。

我们会用 Δv 和 ∇C 重写 ∇C 的变化，即把方程（1）重写为 $\Delta C\approx\nabla C\bullet\Delta v$ .....（4）。

该表达式解释了将 ∇C 称作梯度向量的原因：∇C 把 v 的变化与 C 的变化相关联，正如我们希望的用“梯度”来完成此事。该方程真正的价值在它显示了如何选取 Δv 以使 ΔC 为负，假设我们定：$\Delta v=-\eta\nabla C$ .....(5)，其中 η 是一个很小的正数（称为学习率），由方程（4）可知，$\Delta C\approx-\eta\nabla C\bullet\nabla C=-\eta\|\nabla C\|^2$。由于 $\|\nabla C\|^2\geq 0$，因此 $\Delta C\leq 0$，如果按照方程（5）的规则改变 v，那么 C 会一直减小不会增加（前提是在方程（4）的极限条件下），满足我们的需求。因此，方程（5）用于定义小球在梯度下降算法下的“运动定律”，即可以用方程（5）计算 Δv 的值，根据那个量来移动小球。$v\to v'=v-\eta\nabla C$，然后再次用该更新规则进行下一次移动，反复这样做，C 将不断减小直到到达全局最小值。总之，梯度下降算法的工作方式是重复算梯度 ∇C 然后沿相反方向移动，即沿着山谷滚落。注：Δv 规则只是说：“现在往下。”并非模拟实际物理运动，是抽象而非实际的。以

上讨论的是只有两个变量的函数 C 的梯度下降，实际上即使 C 为多变量函数也能很好工作，假设 C 是一个有 m 个变量$(v_1,...v_m)$的多元函数，那么对应 C 中自变量的变化$\Delta v=(v_1,...v_m)^T$，只需要把上述式子里描述∇C的方程（3）改为$\nabla C\equiv\left(\frac{\partial C}{\partial v_1}\Delta v_1,...,\frac{\partial C}{\partial v_m}\Delta v_m\right)^T$即可。

随机梯度下降不是我们要介绍的重点，使用它是为了加速神经网络学习，思想是通过随机选取少量的训练输入样本来计算每个训练样本的单独损失∇C_x，进而估算平均值∇C，免得求所有单个训练样本的总平均损失浪费太多运算时间。

反向传播的四个方程（BP1-BP4）：

现在回到神经网络的反向传播。思想是利用梯度下降法寻找权重 w 和偏置 b 使损失函数最小。二次代价函数形式如下：$C=\frac{1}{2n}\sum_x\left\|y(x)-a^L(x)\right\|^2$，n 是训练样本总数，求和运算遍历了训练样本 x，y=y(x)是对应目标输出，L 表示神经网络层数，$a^L=a^L(x)$是当输入为 x 时神经网络输出的激活量。**定义**b^l是层 l 的偏置向量，其分量b^l_j为第 l 层第 j 个神经元的偏置，a^l是层 l 的激活向量，其分量a^l_j为第 l 层第 j 个神经元的激活值，即经过 sigmoid 函数的输出值，w^l是第 l 层神经元的权重矩阵，其第 j 行第 k 列的元素是w^l_{jk}。

向量化：对向量 v 中每个元素应用函数，如σ。我们暂且把σ描述为使用了 sigmoid 函数。
那么前文讨论的正向传播 FP 过程可以写为：

$a^l=\sigma(w^la^{l-1}+b^l)$。

提到的中间向量$z^l\equiv w^la^{l-1}+b^l$

z^l的分量是$z^l_j=\sum_k w^l_{jk}a^{l-1}_k+b^l_{jk}$，$z^l_j$是第 l 层第 j 个神经元的激活函数的带权输入。

我们为使用损失函数做出的假设：1.假设损失函数可以写成在每个训练样本上的损失函数C_x的均值，即$C=C_x$。对其中每个单独的训练样本其代价是$C=\frac{1}{2}\left\|y-a^L\right\|^2$。反向传播实际上对单独的训练样本计算了$\frac{\partial C_x}{\partial w}$和$\frac{\partial C_x}{\partial b}$，然后在所有训练样本上进行平均得到$\frac{\partial C}{\partial w}$和$\frac{\partial C}{\partial b}$。这样假设便于丢掉下标，即$C=C_x$。2.假设损失函数可以写成神经网络输出的函数。

介绍一个数学计算符号舒尔积：$\begin{bmatrix}1\\2\end{bmatrix}\odot\begin{bmatrix}3\\4\end{bmatrix}=\begin{bmatrix}1*3\\2*4\end{bmatrix}=\begin{bmatrix}3\\8\end{bmatrix}$

一，假设当每次输入来时，l 层第 j 个神经元上会有微小误差 δ_j^l 存在，导致损失函数的计算结果发生改变。$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$

关于输出层误差的方程，δ_j^L 分量表示为: $\delta_j^L \equiv \frac{\partial C}{\partial a_j^L}\sigma'(z_l^L)$，这是个分量式子，我们把它改成矩阵式：$\delta^L \equiv \nabla_a C \odot \sigma'(z_l^L)$（BP1）

关于二次代价函数有 $\nabla_a C = (a^L - y)$，所以也可以写成 $\delta^L \equiv (a^L - y) \odot \sigma'(z_l^L)$（BP1a）方便用 Numpy 计算。

二，使用下一层的误差 δ^{l+1} 来表示当前层的误差 $\frac{\partial C}{\partial w} = a_{in}\delta_{out}$，有：$\delta^l \equiv ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z_l^L)$（BP2）

当使用 $(w^{l+1})^T$ 时可以把它看作在沿着神经网络反向移动的误差，以此度量第 l 层输出的误差，然后算舒尔积 $\odot\sigma'(z_l^L)$，这会让误差通过第 l 层的激活函数反向传播回来并给出第 l 层的带权输入误差 δ^l。通过组合 BP1，BP2，可以计算任何层的误差 δ^l。首先使用方程 BP1 算 δ^L，再使用 BP2 计算 δ^{L-1}，再用 BP2 计算 δ^{L-2}，如此一步步在神经网络中反向传播。

三，对神经网络的任意偏置，代价函数的变化率如下：$\frac{\partial C}{\partial b_j^l} = \delta_j^l$（BP3），即误差变化率 δ_j^l 与 $\frac{\partial C}{\partial b_j^l}$ 完全一致。由于有 BP1 和 BP2，我们 BP3 可以简写为针对某一个神经元的形式：$\frac{\partial C}{\partial b} = \delta$。

四，对神经网络中的任意权重，代价函数的变化率如下：$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$（BP4）可以简写为：$\frac{\partial C}{\partial w} = a_{in}\delta_{out}$。

a_{in} 是输入到权重 w 的神经元的激活值，δ_{out} 是权重 w 输出的神经元的误差。小激活值的神经元的权重学习会非常缓慢，受梯度影响不大。

反向传播算法：

根据上面的反向传播方程可以计算损失函数的梯度，以算法表达如下：

（1） 输入 x:为输入层设置对应的激活值 a^l。

（2） 向前传播：对每个 l=2,3,…,L,计算相应的 $z^l = w^l a^{l-1} + b^l$ 和 $a^l = \sigma(z^l)$。

（3） 输出层误差 δ^L：计算向量 $\delta^L \equiv \nabla_a C \odot \sigma'(z_l^L)$。

（4） 反向误差传播：对每个 l=L-1,L-2,...,2,计算 $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z_l^L)$。

（5） 输出：代价函数的梯度由 $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ 和 $\frac{\partial C}{\partial b_j^l} = \delta_j^l$ 看出。

已知 $C = C_x$。对于给定大小为 m 的小批量样本，采用如下梯度下降学习（应用）：

（1） 输入训练样本集合。

（2） 对每个训练样本 x,设置对应的输入激活值 $a^{x,l}$，并执行以下步骤：

1. 向前传播：对每个 l=2,3,...,L,计算 $z^{x,l} = w^l a^{x,l-1} + b^l$ 和 $a^{x,l} = \sigma(z^{x,l})$。
2. 输出误差 $\delta^{x,L}$：计算向量 $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z_l^{x,L})$。
3. 反向传播误差：对每个 l=L-1,L-2,...,2,计算 $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z_l^{x,L})$。
4. 梯度下降：对每个 l=L-1,L-2,...,2,根据 $w^l \to w^l - \frac{\eta}{m}\sum_x \delta^{x,l+1}(a^{x,l-1})^T$ 和 $b^l \to b^l - \frac{\eta}{m}\sum_x \delta^{x,l}$ 更新权重和偏置。

在实践中为实现随机梯度下降还需要一个循环来生成小批量训练样本，以及多轮外循环。
反向传播代码：

```
1.  import numpy as np
2.  import  random
3.  class Network(object):
4.      def __init__(self,sizes):
5.          self.num_layers = len(sizes)
6.          self.sizes = sizes
7.          self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
8.          self.weights =[np.random.randn(y,x) for x, y in zip(sizes[:-1], sizes
    [1:])]
9.      #如果第一层 2 个神经元，第二层 3 个，最后层 1 个，初始化为 net = Network([2,3,1)]
10.
11.     def sigmoid(z):
12.         return 1.0/(1.0+np.exp(-z))
13.
14.     def sigmoid_prime(z):
15.         return sigmoid(z)*(1-sigmoid(z))
16.
17.
18.
19.     def update_mini_batch(self,mini_batch,eta):
```

```
        """对一个小批量样本应用梯度下降算法和反向传播算法来更新神经网络的权重和偏置。
        mini_batch 是由若干元组（x,y)组成的列表，eta 是学习率"""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x,y in mini_batch:
            delta_nabla_b, delta_nabla_w, = self.backprop(x, y) #backprop 方法计算偏导数
            nabla_b = [nb+dnb for nb,dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw,dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases,nabla_b)]

def backpop(self, x, y):
    """返回一个表示损失函数 C_x 梯度的元组（nabla_b，nabla_w）。
    nabla_b 和 nabla_w 是一层接一层的 Numpy 数组的列表，类似于 self.biases 和 self.weights."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    #前馈
    activation = x
    activations = [x] #一层接一层地存放所有激活值
    zs = [] #一层接一层地存放所有 z 向量
    for b, w in zip(self.biases, self.weights):
    z = np.dot(w, activation)+b
    zs.append(z)
    activations.append(activation)
    #反向传播
    delta = self.cost_derivate(activations[-1], y) * \
            sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    """以下循环中，l=1 表示最后一层神经元，l=2 表示最后一层，利用了 Python 负索引功能"""
    for l in xrange(2, self.num_layers):
        z = zs[-1]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose, delta) * sp
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b,nabla_w)

    def cost_derivative(self,output_activations, y):
```

```
61.         """返回关于输出激活值的怕偏导数的向量"""
62.         return (output_activations-y)
```