hw4

May 4, 2022

1 COMPSCI-589 HW4: Neural Network

name: Harry (Haochen) Wang

[ ]:
```python
from evaluationmatrix import *
from utils import *
from neuralnetwork import *
from stratified import *
from run import *
import matplotlib.pyplot as plt
```

[ ]:
```python
housedata, housecategory = importhousedata()
winedata, winecategory = importwinedata()
cancerdata, cancercategory = importcancerdata()
cmcdata,cmccategory = importcmcdata()
```

1.0.1 EXAMPLES

place holder for new page.

[ ]:
```python
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
```

backprop_example

May 4, 2022

0.1 EXAMPLES

0.1.1 How to use this?

It's strongly recommend to use the ***backprop_example.ipynb*** file to check the correctness of two examples..

I provide all solutions from the txt file in the #comment, and my code include the print the solutions.

It's also okay to just run the ***example.py*** file, the output might be slightly messy, but it contains all the information needed.

Even though I include most function import from utils, neuralnetwork, etc, it's still recommend to have download all files.

0.1.2 Back Propagation Example 1

[ ]:
```
import numpy as np
from utils import *
from stratified import *
from neuralnetwork import *
```

Forward Propagate

[ ]:
```
def g(x): # sigmoid function
    return 1/(1 + np.exp(-x))
```

[ ]:
```
# Theta 1
# 0.40000  0.10000
# 0.30000  0.20000
theta1 = np.array([[0.4, 0.1],[0.3,0.2]])
# Theta 2
#   0.70000  0.50000  0.60000
theta2 = np.array([0.7,0.5,0.6])
weightlist1= [theta1,theta2]
```

[ ]:
```
# Training set
#         Training instance 1
#                 x: [0.13000]
#                 y: [0.90000]
```

```
#         Training instance 2
#                 x: [0.42000]
#                 y: [0.23000]
# Training instance 1
trainingcategory = {'x1':'numerical', 'y':'class_numerical'}
trainingdata1 = np.array([0.13,0.9])
trainingdata2 = np.array([0.42,0.23])
inputdata1 = np.append(1,trainingdata1[0])
inputdata2 = np.append(1,trainingdata2[0])
exceptout1 = trainingdata1[1]
exceptout2 = trainingdata2[1]
lambda1 = 0
```

[ ]:
```
def costfunction(expected_output, actual_output):
    j = -np.multiply(expected_output,np.log(actual_output)) - np.multiply((1 -␣
 ↪expected_output),np.log(1 - actual_output))
    return np.sum(j)
```

[ ]:
```
def forwardtest(inputdata,weightl,expectedout):
    current_layer_a = inputdata
    print('current_a at 1 is',current_layer_a)
    current_layer_index = 0
    alist = []
    alist.append(current_layer_a)
    for theta in weightl:
        z = np.dot(theta,current_layer_a)
        a = g(z)
        current_layer_a = np.append(1,a) if (current_layer_index+1 !=␣
 ↪len(weightl)) else a
        print('current_a at',current_layer_index+2,'is',current_layer_a)
        alist.append(current_layer_a)
        current_layer_index += 1
    result = current_layer_a
    print('prediction is', result)
    print('exceptout is', expectedout)
    print('cost is', costfunction(expectedout,result))
    return result, costfunction(expectedout,result), alist
```

[ ]:
```
r1,j1,a1 = forwardtest(inputdata1,weightlist1,exceptout1)
# Computing the error/cost, J, of the network
#         Processing training instance 1
#         Forward propagating the input [0.13000]
#                 a1: [1.00000   0.13000]

#                 z2: [0.41300   0.32600]
#                 a2: [1.00000   0.60181   0.58079]
```

```
#                 z3: [1.34937]
#                 a3: [0.79403]

#                 f(x): [0.79403]
#         Predicted output for instance 1: [0.79403]
#         Expected output for instance 1: [0.90000]
#         Cost, J, associated with instance 1: 0.366
```

```
current_a at 1 is [1.   0.13]
current_a at 2 is [1.        0.601807  0.5807858]
current_a at 3 is 0.7940274264318581
prediction is 0.7940274264318581
exceptout is 0.9
cost is 0.36557477431084995
```

[ ]:
```
r2,j2,a2 = forwardtest(inputdata2,weightlist1,exceptout2)
        # Processing training instance 2
        # Forward propagating the input [0.42000]
        #         a1: [1.00000   0.42000]

        #         z2: [0.44200   0.38400]
        #         a2: [1.00000   0.60874   0.59484]

        #         z3: [1.36127]
        #         a3: [0.79597]

        #         f(x): [0.79597]
        # Predicted output for instance 2: [0.79597]
        # Expected output for instance 2: [0.23000]
        # Cost, J, associated with instance 2: 1.276
```

```
current_a at 1 is [1.   0.42]
current_a at 2 is [1.         0.60873549 0.59483749]
current_a at 3 is 0.7959660671522611
prediction is 0.7959660671522611
exceptout is 0.23
cost is 1.2763768066887786
```

[ ]:
```
jlist1 = np.array([j1,j2])
numberofinstance1 = 2
```

[ ]:
```
def overallcost(jlist,n,weightl,lambda_reg):
    s = sumofweights(weightl,bias=0)*lambda_reg/(2*n)
    jsum = np.sum(jlist)
    return jsum/n + s
```

[ ]:
```
overallcost(jlist1,numberofinstance1,weightlist1,lambda1)
# Final (regularized) cost, J, based on the complete training set: 0.82098
```

[ ]: 0.8209757904998143

Back Propagate

```
[ ]: def delta(weightl,alist,expect,actual):
         delta_layer_n = actual-expect
         deltalist = []
         deltalist.append(delta_layer_n)
         i = len(weightl)-1
         current_delta = delta_layer_n
         while i > 0:
             delta_layer_now = np.multiply(np.multiply(np.dot(weightl[i].
      ↪T,current_delta),alist[i]),(1-alist[i]))
             current_delta = delta_layer_now[1:]
             deltalist.append(current_delta)
             i-=1
         deltalist.reverse()
         return deltalist
```

```
[ ]: def gradientD(weights_list,delta_list,a_list,biasterm=True):
         gradlist = []
         for i in range(len(weights_list)):
             anow = a_list[i]
             deltanow = np.array([delta_list[i]]).T
             dotproduct = deltanow*anow
             # print('dotshape',dotproduct.shape)
             gradlist.append(dotproduct)
         return gradlist
```

```
[ ]: delta1_1 = delta(weightlist1,a1,exceptout1,r1)
             # Computing gradients based on training instance 1
             #           delta3: [-0.10597]
             #           delta2: [-0.01270    -0.01548]
     print(delta1_1)
```

```
[array([-0.01269739, -0.01548092]), -0.10597257356814194]
```

```
[ ]:                 # Gradients of Theta2 based on training instance 1:
                     #           -0.10597  -0.06378  -0.06155

                     # Gradients of Theta1 based on training instance 1:
                     #           -0.01270  -0.00165
                     #           -0.01548  -0.00201
     gradd1_1 = gradientD(weightlist1,delta1_1,a1)
     print(gradd1_1)
```

```
[array([[-0.01269739, -0.00165066],
       [-0.01548092, -0.00201252]]), array([-0.10597257, -0.06377504,
-0.06154737])]
```

[ ]:
```python
delta1_2 = delta(weightlist1,a2,exceptout2,r2)
        # Computing gradients based on training instance 2
        #         delta3: [0.56597]
        #         delta2: [0.06740   0.08184]
print(delta1_2)
```

```
[array([0.06739994, 0.08184068]), 0.5659660671522612]
```

[ ]:
```python
                # Gradients of Theta2 based on training instance 2:
                #         0.56597  0.34452  0.33666

                # Gradients of Theta1 based on training instance 2:
                #         0.06740  0.02831
                #         0.08184  0.03437
gradd1_2 = gradientD(weightlist1,delta1_2,a2)
print(gradd1_2)
```

```
[array([[0.06739994, 0.02830797],
       [0.08184068, 0.03437309]]), array([0.56596607, 0.34452363, 0.33665784])]
```

[ ]:
```python
def transposelistoflist(l):
    newlistoflist = []
    for i in range(len(l[0])):
        newlist = []
        for j in range(len(l)):
            newlist.append(l[j][i])
        newlistoflist.append(newlist)
    return newlistoflist
```

[ ]:
```python
listofgradient = [gradd1_1,gradd1_2]
gradientP1 = [lambda1*t for t in weightlist1]
grad_D_transpose = transposelistoflist(listofgradient)
grad_D_sum = [np.sum(t,axis=0) for t in grad_D_transpose]
update_gradients = []
for i in range(len(grad_D_sum)):
    update_gradients.append((grad_D_sum[i] + gradientP1[i])*(1/
 ↪numberofinstance1))
```

[ ]:
```python
print(update_gradients)
        # The entire training set has been processes. Computing the average␣
 ↪(regularized) gradients:
        #         Final regularized gradients of Theta1:
        #                 0.02735  0.01333
        #                 0.03318  0.01618

        #         Final regularized gradients of Theta2:
        #                 0.23000  0.14037  0.13756
```

```
[array([[0.02735127, 0.01332866],
```

```
[0.03317988, 0.01618028]]), array([0.22999675, 0.1403743 , 0.13755523])]
```

0.1.3 Back Propagation Example 2

Forward Propagrate

```
[ ]: # Initial Theta1 (the weights of each neuron, including the bias weight, are
     ↪stored in the rows):
     #          0.42000  0.15000  0.40000
     #          0.72000  0.10000  0.54000
     #          0.01000  0.19000  0.42000
     #          0.30000  0.35000  0.68000

     # Initial Theta2 (the weights of each neuron, including the bias weight, are
     ↪stored in the rows):
     #          0.21000  0.67000  0.14000  0.96000  0.87000
     #          0.87000  0.42000  0.20000  0.32000  0.89000
     #          0.03000  0.56000  0.80000  0.69000  0.09000

     # Initial Theta3 (the weights of each neuron, including the bias weight, are
     ↪stored in the rows):
     #          0.04000  0.87000  0.42000  0.53000
     #          0.17000  0.10000  0.95000  0.69000
     e2theta1 = np.array([[0.42,0.15,0.4],[0.72,0.1,0.54],[0.01,0.19,0.42],[0.3,0.
     ↪35,0.68]])
     e2theta2 = np.array([[0.21,0.67,0.14,0.96,0.87],[0.87,0.42,0.2,0.32,0.89],[0.
     ↪03,0.56,0.8,0.69,0.09]])
     e2theta3 = np.array([[0.04,0.87,0.42,0.53],[0.17,0.1,0.95,0.69]])
     e2weightlist = [e2theta1,e2theta2,e2theta3]
```

```
[ ]: # Training set
     #          Training instance 1
     #                  x: [0.32000   0.68000]
     #                  y: [0.75000   0.98000]
     #          Training instance 2
     #                  x: [0.83000   0.02000]
     #                  y: [0.75000   0.28000]

     e2input1 = np.array([0.32,0.68])
     e2input2 = np.array([0.83,0.02])
     e2exceptout1 = np.array([0.75,0.98])
     e2exceptout2 = np.array([0.75,0.28])

     e2input1 = np.append(1,e2input1)
     e2input2 = np.append(1,e2input2)
     e2lambda0 = 0.25
```

[ ]:
```
e2r1,e2j1,e2a1 = forwardtest(e2input1,e2weightlist,e2exceptout1)
        # Processing training instance 1
        # Forward propagating the input [0.32000   0.68000]
        #         a1: [1.00000   0.32000   0.68000]

        #         z2: [0.74000   1.11920   0.35640   0.87440]
        #         a2: [1.00000   0.67700   0.75384   0.58817   0.70566]

        #         z3: [1.94769   2.12136   1.48154]
        #         a3: [1.00000   0.87519   0.89296   0.81480]

        #         z4: [1.60831   1.66805]
        #         a4: [0.83318   0.84132]

        #         f(x): [0.83318   0.84132]
        # Predicted output for instance 1: [0.83318   0.84132]
        # Expected output for instance 1: [0.75000   0.98000]
        # Cost, J, associated with instance 1: 0.791
```

```
current_a at 1 is [1.   0.32 0.68]
current_a at 2 is [1.         0.67699586 0.75384029 0.5881687  0.70566042]
current_a at 3 is [1.         0.87519469 0.89296181 0.81480444]
current_a at 4 is [0.83317658 0.84131543]
prediction is [0.83317658 0.84131543]
exceptout is [0.75 0.98]
cost is 0.7907366961135718
```

[ ]:
```
e2r2,e2j2,e2a2 = forwardtest(e2input2,e2weightlist,e2exceptout2)
        # Processing training instance 2
        # Forward propagating the input [0.83000   0.02000]
        #         a1: [1.00000   0.83000   0.02000]

        #         z2: [0.55250   0.81380   0.17610   0.60410]
        #         a2: [1.00000   0.63472   0.69292   0.54391   0.64659]

        #         z3: [1.81696   2.02468   1.37327]
        #         a3: [1.00000   0.86020   0.88336   0.79791]

        #         z4: [1.58228   1.64577]
        #         a4: [0.82953   0.83832]

        #         f(x): [0.82953   0.83832]
        # Predicted output for instance 2: [0.82953   0.83832]
        # Expected output for instance 2: [0.75000   0.28000]
        # Cost, J, associated with instance 2: 1.944
```

```
current_a at 1 is [1.   0.83 0.02]
current_a at 2 is [1.         0.63471542 0.69291867 0.54391158 0.64659376]
```

```
current_a at 3 is [1.         0.86020091 0.88336451 0.79790763]
current_a at 4 is [0.82952703 0.83831889]
prediction is [0.82952703 0.83831889]
exceptout is [0.75 0.28]
cost is 1.9437823352945296
```

[ ]:
```
e2jlist = np.array([e2j1,e2j2])
e2numberofinstance = 2
```

[ ]:
```
def sumofweights(listofweights,bias=True): # computes the square of all weights␣
↪of the network and sum them up
    sum = 0
    for weight in listofweights:
        if bias:
            w = weight.copy()
            w[:, 0] = 0
            sum += np.sum(np.square(w))
        else:
            sum += np.sum(np.square(weight))
    return sum
```

[ ]:
```
def overallcost(jlist,n,weightl,lambda_reg):
    s = sumofweights(weightl,bias=1)*lambda_reg/(2*n)
    jsum = np.sum(jlist)
    return jsum/n + s
```

[ ]:
```
overallcost(e2jlist,e2numberofinstance,e2weightlist,e2lambda0)
# Final (regularized) cost, J, based on the complete training set: 1.90351
```

[ ]: 1.9035095157040507

Back Propagation for E2

[ ]:
```
e2delta1 = delta(e2weightlist,e2a1,e2exceptout1,e2r1)
# Running backpropagation
#         Computing gradients based on training instance 1
#                 delta4: [0.08318   -0.13868]
#                 delta3: [0.00639   -0.00925   -0.00779]
#                 delta2: [-0.00087   -0.00133   -0.00053   -0.00070]
print(e2delta1)
```

```
[array([-0.00086743, -0.00133354, -0.00053312, -0.00070163]), array([
0.00638937, -0.00925379, -0.00778767]), array([ 0.08317658, -0.13868457])]
```

[ ]:
```
                # Gradients of Theta3 based on training instance 1:
                #         0.08318  0.07280  0.07427  0.06777
                #         -0.13868  -0.12138  -0.12384  -0.11300

                # Gradients of Theta2 based on training instance 1:
```

```
                #       0.00639  0.00433  0.00482  0.00376  0.00451
                #       -0.00925  -0.00626  -0.00698  -0.00544  -0.00653
                #       -0.00779  -0.00527  -0.00587  -0.00458  -0.00550

                # Gradients of Theta1 based on training instance 1:
                #       -0.00087  -0.00028  -0.00059
                #       -0.00133  -0.00043  -0.00091
                #       -0.00053  -0.00017  -0.00036
                #       -0.00070  -0.00022  -0.00048
e2grad1 = gradientD(e2weightlist,e2delta1,e2a1)
print(e2grad1)
```

```
[array([[-0.00086743, -0.00027758, -0.00058985],
       [-0.00133354, -0.00042673, -0.00090681],
       [-0.00053312, -0.0001706 , -0.00036252],
       [-0.00070163, -0.00022452, -0.00047711]]), array([[ 0.00638937,
0.00432557,  0.00481656,  0.00375802,  0.00450872],
       [-0.00925379, -0.00626478, -0.00697588, -0.00544279, -0.00653003],
       [-0.00778767, -0.00527222, -0.00587066, -0.00458046, -0.00549545]]),
array([[ 0.08317658,  0.0727957 ,  0.07427351,  0.06777264],
       [-0.13868457, -0.121376  , -0.12384003, -0.1130008 ]])]
```

[ ]:
```
e2delta2 = delta(e2weightlist,e2a2,e2exceptout2,e2r2)
        # Computing gradients based on training instance 2
        #       delta4: [0.07953   0.55832]
        #       delta3: [0.01503   0.05809   0.06892]
        #       delta2: [0.01694   0.01465   0.01999   0.01622]
print(e2delta2)
```

```
[array([0.01694006, 0.01465141, 0.01998824, 0.01622017]), array([0.01503437,
0.05808969, 0.06891698]), array([0.07952703, 0.55831889])]
```

[ ]:
```
                # Gradients of Theta3 based on training instance 2:
                #       0.07953  0.06841  0.07025  0.06346
                #       0.55832  0.48027  0.49320  0.44549

                # Gradients of Theta2 based on training instance 2:
                #       0.01503  0.00954  0.01042  0.00818  0.00972
                #       0.05809  0.03687  0.04025  0.03160  0.03756
                #       0.06892  0.04374  0.04775  0.03748  0.04456

                # Gradients of Theta1 based on training instance 2:
                #       0.01694  0.01406  0.00034
                #       0.01465  0.01216  0.00029
                #       0.01999  0.01659  0.00040
                #       0.01622  0.01346  0.00032
e2grad2 = gradientD(e2weightlist,e2delta2,e2a2)
print(e2grad2)
```

```
[array([[0.01694006, 0.01406025, 0.0003388 ],
       [0.01465141, 0.01216067, 0.00029303],
       [0.01998824, 0.01659024, 0.00039976],
       [0.01622017, 0.01346274, 0.0003244 ]]), array([[0.01503437, 0.00954254,
0.01041759, 0.00817737, 0.00972113],
       [0.05808969, 0.03687042, 0.04025143, 0.03159565, 0.03756043],
       [0.06891698, 0.04374267, 0.04775386, 0.03748474, 0.04456129]]),
array([[0.07952703, 0.06840922, 0.07025135, 0.06345522],
       [0.55831889, 0.48026642, 0.4931991 , 0.44548691]])]
```

```
[ ]: e2listofgradient = [e2grad1,e2grad2]
gradientP2 = [e2lambda0*t for t in e2weightlist]
for singleP in gradientP2:
    singleP[:, 0] = 0
e2_grad_D_transpose = transposelistoflist(e2listofgradient)
e2_grad_D_sum = [np.sum(t,axis=0) for t in e2_grad_D_transpose]
e2_update_gradients = []
for i in range(len(grad_D_sum)):
    e2_update_gradients.append((e2_grad_D_sum[i] + gradientP2[i])*(1/
 ↪e2numberofinstance))
```

```
[ ]: print(e2_update_gradients)
        # The entire training set has been processes. Computing the average␣
 ↪(regularized) gradients:
        #         Final regularized gradients of Theta1:
        #                 0.00804  0.02564  0.04987
        #                 0.00666  0.01837  0.06719
        #                 0.00973  0.03196  0.05252
        #                 0.00776  0.05037  0.08492

        #         Final regularized gradients of Theta2:
        #                 0.01071  0.09068  0.02512  0.12597  0.11586
        #                 0.02442  0.06780  0.04164  0.05308  0.12677
        #                 0.03056  0.08924  0.12094  0.10270  0.03078

        #         Final regularized gradients of Theta3:
        #                 0.08135  0.17935  0.12476  0.13186
        #                 0.20982  0.19195  0.30343  0.25249
```

```
[array([[0.00803632, 0.02564134, 0.04987447],
       [0.00665894, 0.01836697, 0.06719311],
       [0.00972756, 0.03195982, 0.05251862],
       [0.00775927, 0.05036911, 0.08492365]]), array([[0.01071187, 0.09068406,
0.02511708, 0.1259677 , 0.11586492],
       [0.02441795, 0.06780282, 0.04163777, 0.05307643, 0.1267652 ],
       [0.03056466, 0.08923522, 0.1209416 , 0.10270214, 0.03078292]])]
```

```
#
```

1.1 EXPERIMENTS & ANALYSES

1.1.1 House Data

Trained with MiniBatch, vectorized neural network, divided to 3 group, (batchsize = 435/3 = 145)

(I named the minibatchk to the batches I want to divided to.)

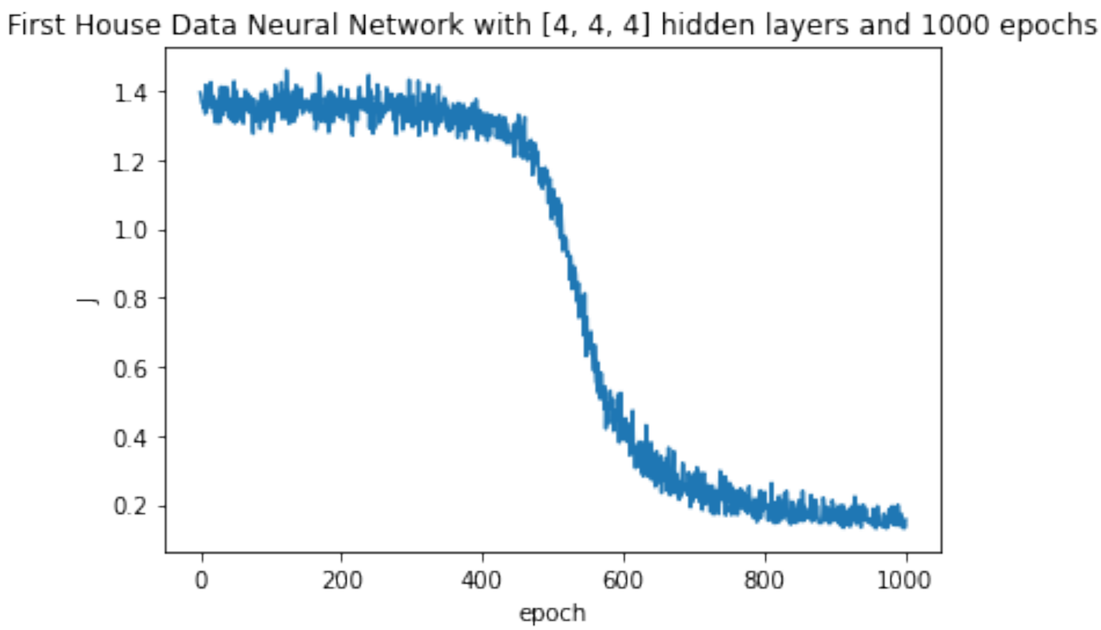
```
[ ]: hiddenlayerparemeter = [4,4,4]
     epoch = 1000
```

```
[ ]: listoflistofoutputs, acc, listofjlist = kfoldcrossvalidneuralnetwork(housedata,␣
     ↪housecategory, hiddenlayerparemeter, k = 5, minibatchk = 3, lambda_reg = 0.
     ↪1, learning_rate = 0.1, epsilon_0 = 0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9204545454545454
fold 2 training in progress
fold 2 training completed, accuracy =  0.9318181818181818
fold 3 training in progress
fold 3 training completed, accuracy =  0.9425287356321839
fold 4 training in progress
fold 4 training completed, accuracy =  0.9883720930232558
fold 5 training in progress
fold 5 training completed, accuracy =  0.9767441860465116
```

```
[ ]: accuarcy_, precision_, recall_, fscore_house_1=␣
     ↪meanevaluation(listoflistofoutputs,1)
     print("First House Data Neural Network with " + str(hiddenlayerparemeter) + "␣
     ↪hidden layers and " + str(epoch) + " epochs")
     print("accuracy:", acc)
     print("fscore:", fscore_house_1)
     plt.plot(range(epoch+1), listofjlist[1])
     plt.xlabel("epoch")
     plt.ylabel("J")
     plt.title("First House Data Neural Network with " + str(hiddenlayerparemeter) +␣
     ↪" hidden layers and " + str(epoch) + " epochs")
     plt.show()
```

```
First House Data Neural Network with [4, 4, 4] hidden layers and 1000 epochs
accuracy: 0.9519835483949357
fscore: 0.9385771973545435
```

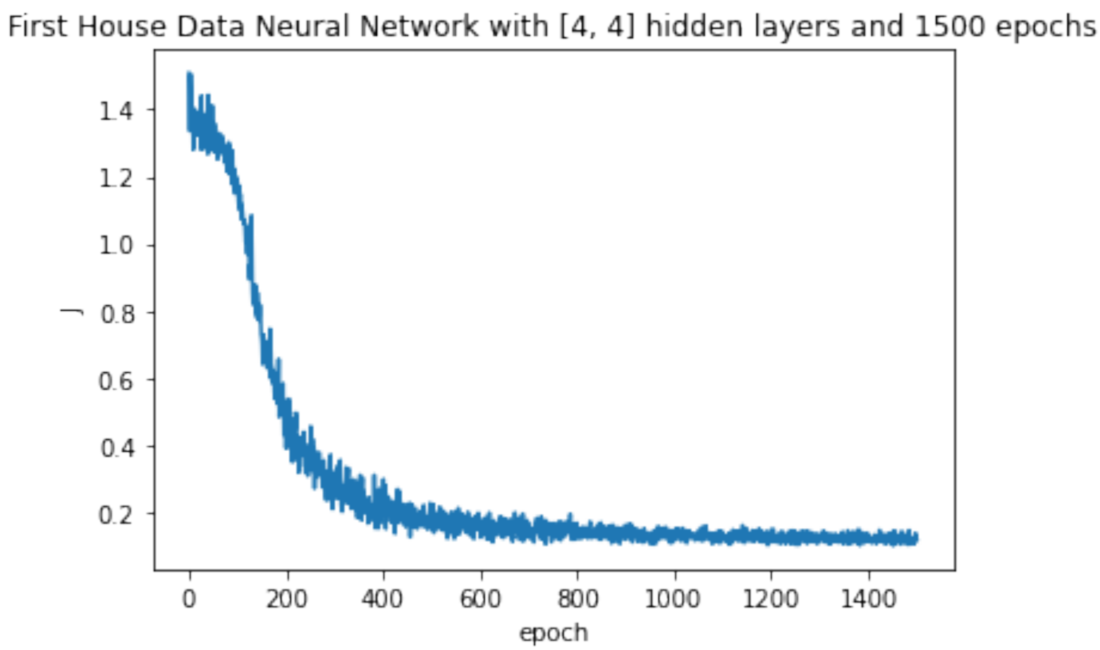



```
[ ]: hiddenlayerparemeter = [4,4]
     epoch = 1500
     listoflistofoutputs1_2, acc1_2, listofjlist1_2 =␣
      ↪kfoldcrossvalidneuralnetwork(housedata, housecategory, hiddenlayerparemeter,␣
      ↪k = 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
      ↪0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  1.0
fold 2 training in progress
fold 2 training completed, accuracy =  0.9090909090909091
fold 3 training in progress
fold 3 training completed, accuracy =  0.9772727272727273
fold 4 training in progress
fold 4 training completed, accuracy =  0.9545454545454546
fold 5 training in progress
fold 5 training completed, accuracy =  0.9318181818181818
fold 6 training in progress
fold 6 training completed, accuracy =  0.9772727272727273
fold 7 training in progress
fold 7 training completed, accuracy =  0.9772727272727273
fold 8 training in progress
fold 8 training completed, accuracy =  1.0
fold 9 training in progress
fold 9 training completed, accuracy =  0.9285714285714286
fold 10 training in progress
fold 10 training completed, accuracy =  0.9761904761904762
```

[ ]:
```
accuarcy1_2, precision1_2, recall1_2, fscore_house1_2=␣
    ↪meanevaluation(listoflistofoutputs1_2,1)
print("First House Data Neural Network with " + str(hiddenlayerparemeter) + "␣
    ↪hidden layers and " + str(epoch) + " epochs")
print("accuracy:", acc1_2)
print("fscore:", fscore_house1_2)
plt.plot(range(epoch+1), listofjlist1_2[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("First House Data Neural Network with " + str(hiddenlayerparemeter) +␣
    ↪" hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
First House Data Neural Network with [4, 4] hidden layers and 1500 epochs
accuracy: 0.9632034632034632
fscore: 0.9513545046562125
```



[ ]:
```
hiddenlayerparemeter = [4,4,4,4]
epoch = 1500
listoflistofoutputs1_3, acc1_3, listofjlist1_3 =␣
    ↪kfoldcrossvalidneuralnetwork(housedata, housecategory, hiddenlayerparemeter,␣
    ↪k = 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
    ↪0001, softstop = epoch, printq = False)
```

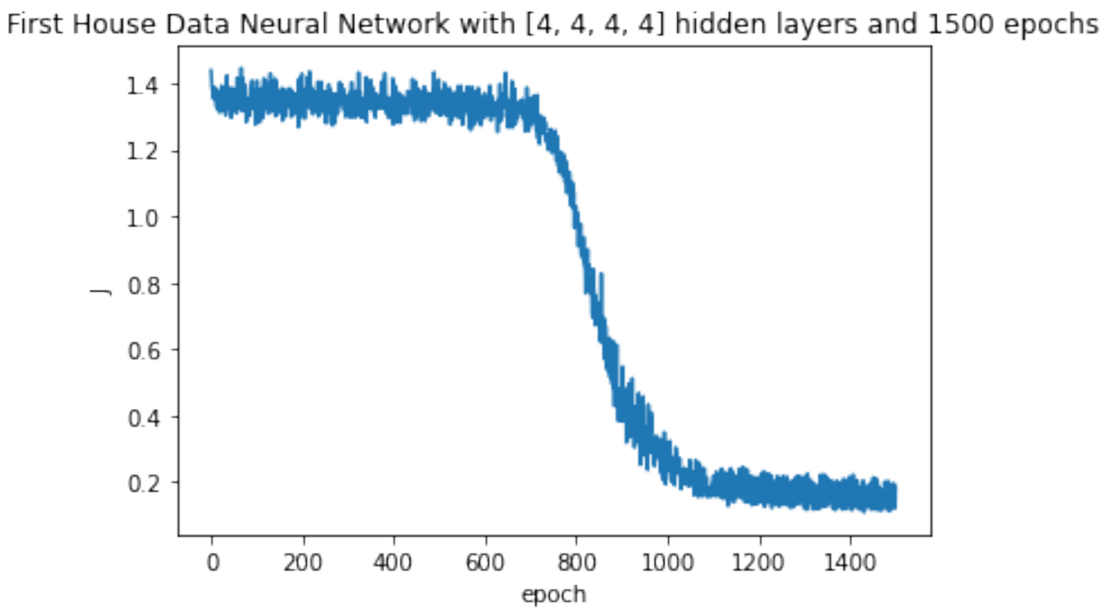
```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9772727272727273
```

```
fold 2 training in progress
fold 2 training completed, accuracy =  0.9545454545454546
fold 3 training in progress
fold 3 training completed, accuracy =  0.8409090909090909
fold 4 training in progress
fold 4 training completed, accuracy =  0.6136363636363636
fold 5 training in progress
fold 5 training completed, accuracy =  0.6136363636363636
fold 6 training in progress
fold 6 training completed, accuracy =  0.6136363636363636
fold 7 training in progress
fold 7 training completed, accuracy =  0.6136363636363636
fold 8 training in progress
fold 8 training completed, accuracy =  0.9767441860465116
fold 9 training in progress
fold 9 training completed, accuracy =  0.6190476190476191
fold 10 training in progress
fold 10 training completed, accuracy =  0.9285714285714286
```

[ ]:

```
accuarcy1_3, precision1_3, recall1_3, fscore_house1_3=␣
↪meanevaluation(listoflistofoutputs1_3,1)
print("First House Data Neural Network with " + str(hiddenlayerparemeter) + "␣
↪hidden layers and " + str(epoch) + " epochs")
print("accuracy:", acc1_3)
print("fscore:", fscore_house1_3)
plt.plot(range(epoch+1), listofjlist1_3[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("First House Data Neural Network with " + str(hiddenlayerparemeter) +␣
↪" hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
First House Data Neural Network with [4, 4, 4, 4] hidden layers and 1500 epochs
accuracy: 0.7751635960938288
fscore: 0.45981240981240984
```

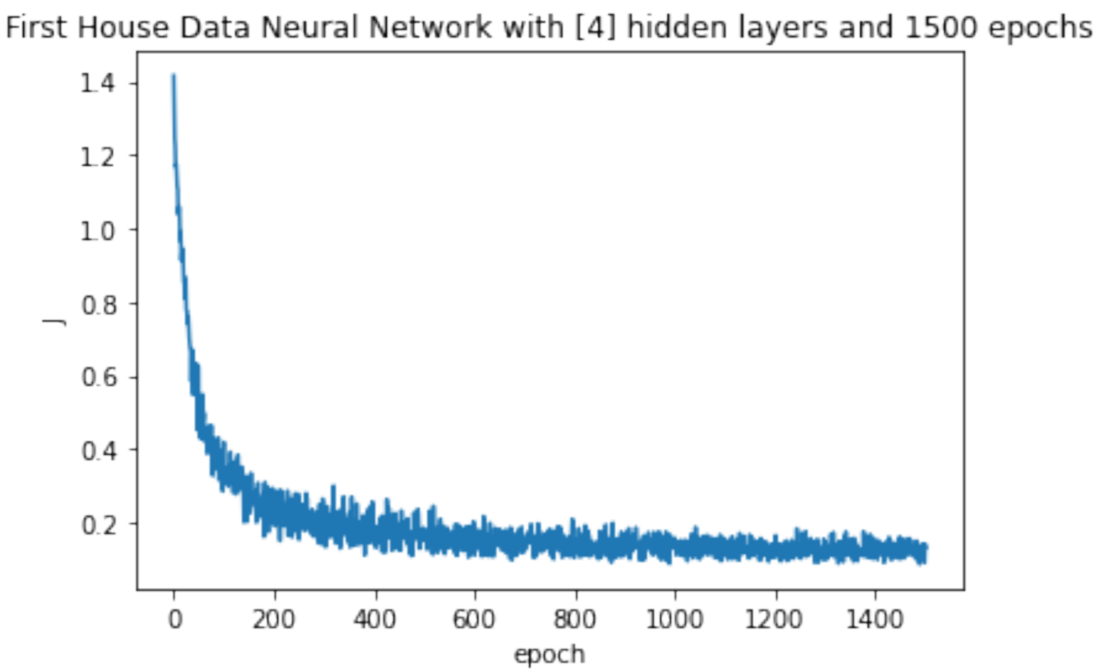



```
[ ]: hiddenlayerparemeter = [4]
     epoch = 1500
     listoflistofoutputs1_4, acc1_4, listofjlist1_4 =␣
      ↪kfoldcrossvalidneuralnetwork(housedata, housecategory, hiddenlayerparemeter,␣
      ↪k = 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
      ↪0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  1.0
fold 2 training in progress
fold 2 training completed, accuracy =  0.9318181818181818
fold 3 training in progress
fold 3 training completed, accuracy =  0.9545454545454546
fold 4 training in progress
fold 4 training completed, accuracy =  0.9772727272727273
fold 5 training in progress
fold 5 training completed, accuracy =  0.9318181818181818
fold 6 training in progress
fold 6 training completed, accuracy =  0.9772727272727273
fold 7 training in progress
fold 7 training completed, accuracy =  0.9318181818181818
fold 8 training in progress
fold 8 training completed, accuracy =  0.9302325581395349
fold 9 training in progress
fold 9 training completed, accuracy =  1.0
fold 10 training in progress
fold 10 training completed, accuracy =  0.9761904761904762
```

[ ]:
```
accuarcy1_4, precision1_4, recall1_4, fscore_house1_4=␣
    ↪meanevaluation(listoflistofoutputs1_4,1)
print("First House Data Neural Network with " + str(hiddenlayerparemeter) + "␣
    ↪hidden layers and " + str(epoch) + " epochs")
print("accuracy:", acc1_4)
print("fscore:", fscore_house1_4)
plt.plot(range(epoch+1), listofjlist1_4[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("First House Data Neural Network with " + str(hiddenlayerparemeter) +␣
    ↪" hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
First House Data Neural Network with [4] hidden layers and 1500 epochs
accuracy: 0.9610968488875467
fscore: 0.9490942256668063
```



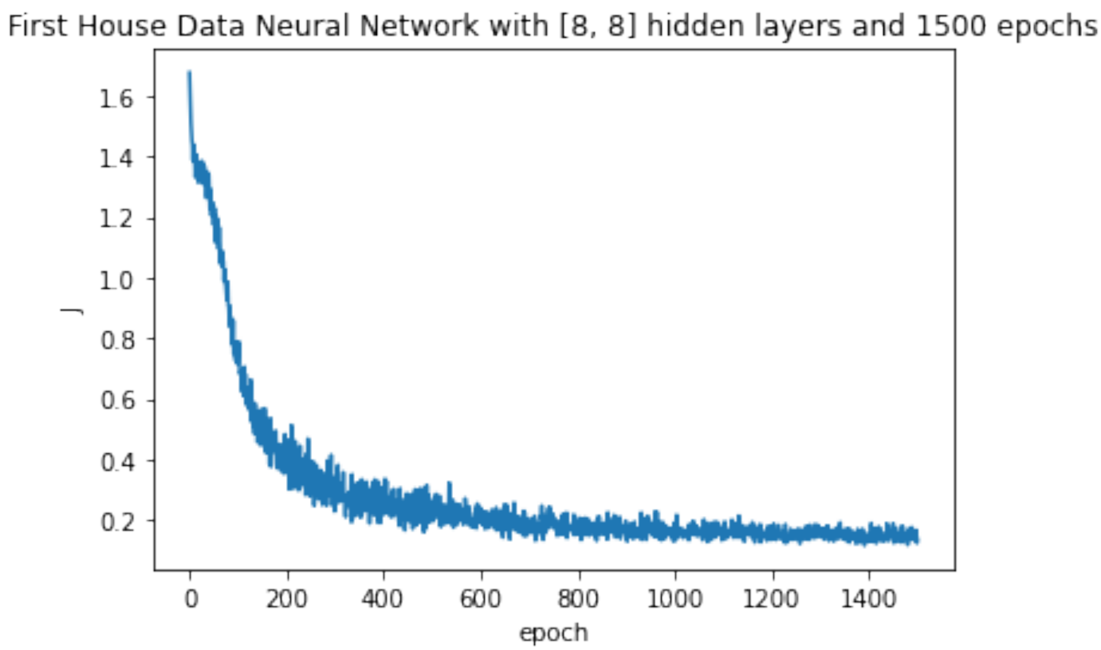
[ ]:
```
hiddenlayerparemeter = [8,8]
epoch = 1500
listoflistofoutputs1_5, acc1_5, listofjlist1_5 =␣
    ↪kfoldcrossvalidneuralnetwork(housedata, housecategory, hiddenlayerparemeter,␣
    ↪k = 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.09, epsilon_0 =␣
    ↪0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9772727272727273
```

```
fold 2 training in progress
fold 2 training completed, accuracy =  1.0
fold 3 training in progress
fold 3 training completed, accuracy =  0.8636363636363636
fold 4 training in progress
fold 4 training completed, accuracy =  0.8636363636363636
fold 5 training in progress
fold 5 training completed, accuracy =  0.9545454545454546
fold 6 training in progress
fold 6 training completed, accuracy =  0.9772727272727273
fold 7 training in progress
fold 7 training completed, accuracy =  0.9318181818181818
fold 8 training in progress
fold 8 training completed, accuracy =  0.9767441860465116
fold 9 training in progress
fold 9 training completed, accuracy =  1.0
fold 10 training in progress
fold 10 training completed, accuracy =  1.0
```

[ ]:
```
accuarcy1_5, precision1_5, recall1_5, fscore_house1_5=␣
↪meanevaluation(listoflistofoutputs1_5,1)
print("First House Data Neural Network with " + str(hiddenlayerparemeter) + "␣
↪hidden layers and " + str(epoch) + " epochs")
print("accuracy:", acc1_5)
print("fscore:", fscore_house1_5)
plt.plot(range(epoch+1), listofjlist1_5[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("First House Data Neural Network with " + str(hiddenlayerparemeter) +␣
↪" hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
First House Data Neural Network with [8, 8] hidden layers and 1500 epochs
accuracy: 0.954492600422833
fscore: 0.9424683812919106
```



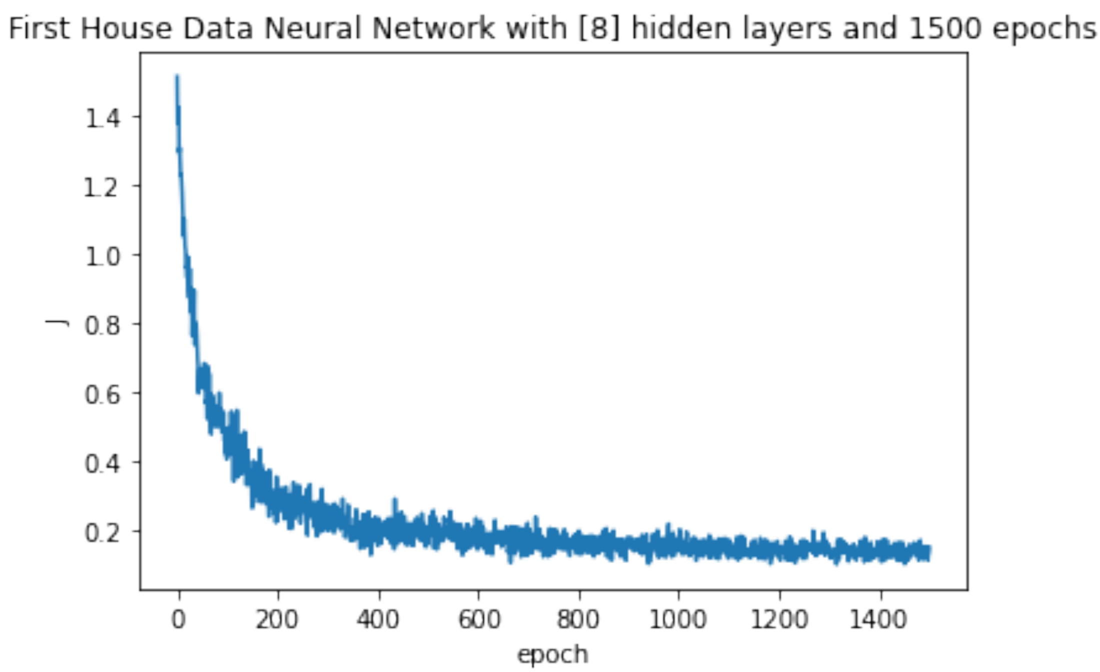
[ ]:
```
hiddenlayerparemeter = [8]
epoch = 1500
listoflistofoutputs1_6, acc1_6, listofjlist1_6 =␣
↪kfoldcrossvalidneuralnetwork(housedata, housecategory, hiddenlayerparemeter,␣
↪k = 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.09, epsilon_0 =␣
↪0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  1.0
fold 2 training in progress
fold 2 training completed, accuracy =  0.9545454545454546
fold 3 training in progress
fold 3 training completed, accuracy =  0.9772727272727273
fold 4 training in progress
fold 4 training completed, accuracy =  0.9318181818181818
fold 5 training in progress
fold 5 training completed, accuracy =  0.9772727272727273
fold 6 training in progress
fold 6 training completed, accuracy =  0.9772727272727273
fold 7 training in progress
fold 7 training completed, accuracy =  0.8863636363636364
fold 8 training in progress
fold 8 training completed, accuracy =  0.9534883720930233
fold 9 training in progress
fold 9 training completed, accuracy =  0.9523809523809523
fold 10 training in progress
```

```
fold 10 training completed, accuracy =  1.0
```

[ ]:
```
accuarcy1_6, precision1_6, recall1_6, fscore_house1_6=␣
↪meanevaluation(listoflistofoutputs1_6,1)
print("First House Data Neural Network with " + str(hiddenlayerparemeter) + "␣
↪hidden layers and " + str(epoch) + " epochs")
print("accuracy:", acc1_6)
print("fscore:", fscore_house1_6)
plt.plot(range(epoch+1), listofjlist1_6[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("First House Data Neural Network with " + str(hiddenlayerparemeter) +␣
↪" hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
First House Data Neural Network with [8] hidden layers and 1500 epochs
accuracy: 0.961041477901943
fscore: 0.9477564986287096
```

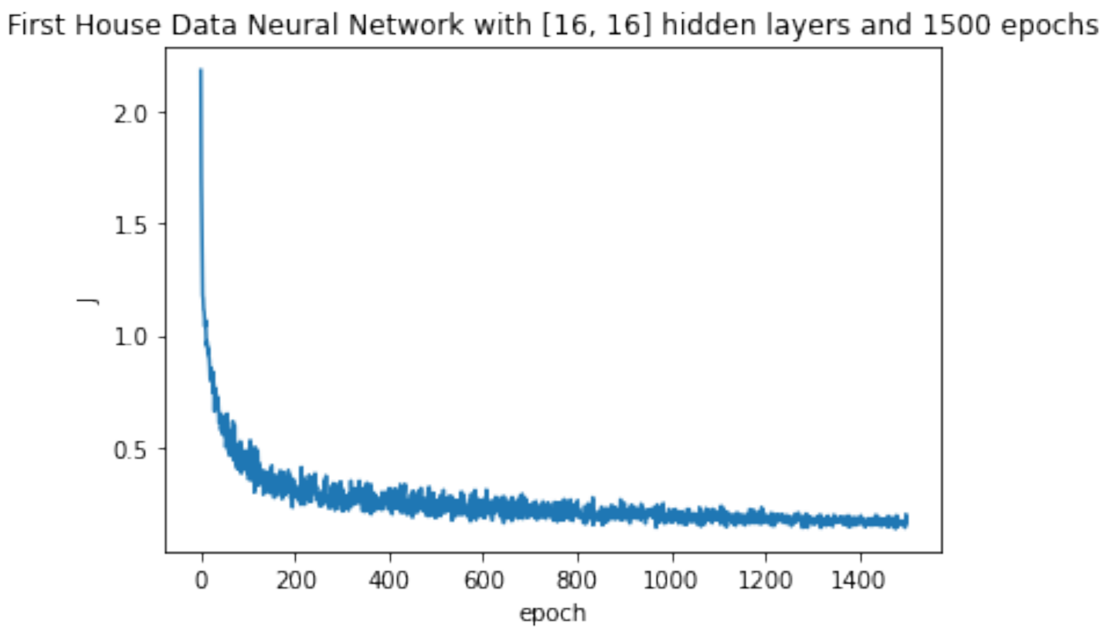


[ ]:
```
hiddenlayerparemeter = [16,16]
epoch = 1500
listoflistofoutputs1_7, acc1_7, listofjlist1_7 =␣
↪kfoldcrossvalidneuralnetwork(housedata, housecategory, hiddenlayerparemeter,␣
↪k = 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.09, epsilon_0 =␣
↪0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9545454545454546
fold 2 training in progress
fold 2 training completed, accuracy =  0.9318181818181818
fold 3 training in progress
fold 3 training completed, accuracy =  0.9545454545454546
fold 4 training in progress
fold 4 training completed, accuracy =  0.9772727272727273
fold 5 training in progress
fold 5 training completed, accuracy =  0.9090909090909091
fold 6 training in progress
fold 6 training completed, accuracy =  0.9772727272727273
fold 7 training in progress
fold 7 training completed, accuracy =  1.0
fold 8 training in progress
fold 8 training completed, accuracy =  0.9302325581395349
fold 9 training in progress
fold 9 training completed, accuracy =  0.9523809523809523
fold 10 training in progress
fold 10 training completed, accuracy =  1.0
```

[ ]:
```
accuarcy1_7, precision1_7, recall1_7, fscore_house1_7=␣
 ↪meanevaluation(listoflistofoutputs1_7,1)
print("First House Data Neural Network with " + str(hiddenlayerparemeter) + "␣
 ↪hidden layers and " + str(epoch) + " epochs")
print("accuracy:", acc1_7)
print("fscore:", fscore_house1_7)
plt.plot(range(epoch+1), listofjlist1_7[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("First House Data Neural Network with " + str(hiddenlayerparemeter) +␣
 ↪" hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
First House Data Neural Network with [16, 16] hidden layers and 1500 epochs
accuracy: 0.9587158965065943
fscore: 0.9472877623612916
```

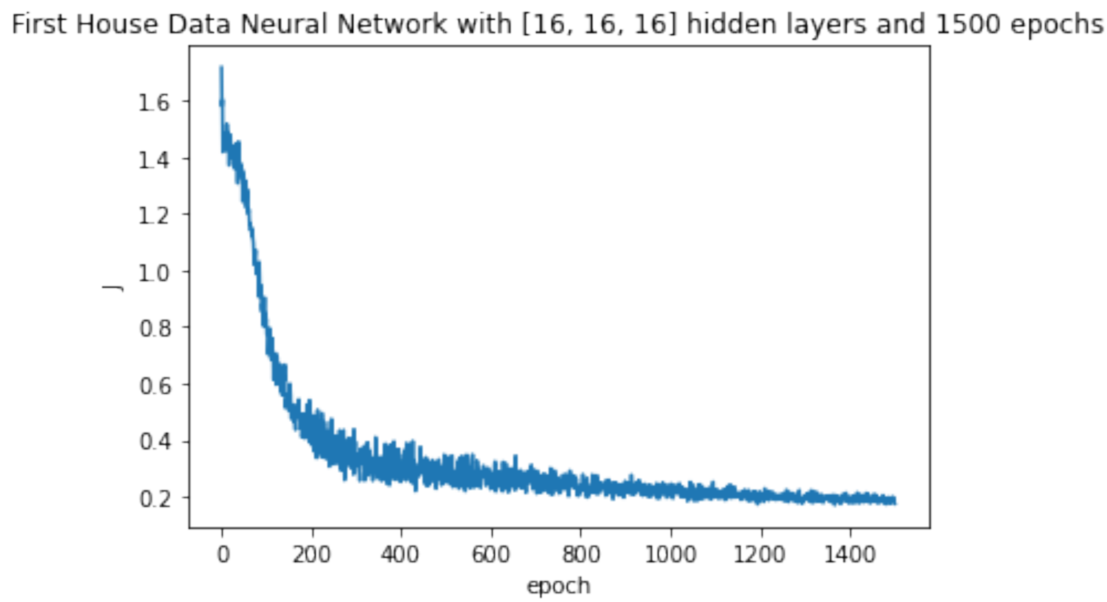


[ ]:
```
hiddenlayerparemeter = [16,16,16]
epoch = 1500
listoflistofoutputs1_8, acc1_8, listofjlist1_8 = 
 ↪kfoldcrossvalidneuralnetwork(housedata, housecategory, hiddenlayerparemeter, 
 ↪k = 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.09, epsilon_0 = 
 ↪0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9318181818181818
fold 2 training in progress
fold 2 training completed, accuracy =  0.9545454545454546
fold 3 training in progress
fold 3 training completed, accuracy =  1.0
fold 4 training in progress
fold 4 training completed, accuracy =  0.9545454545454546
fold 5 training in progress
fold 5 training completed, accuracy =  0.9545454545454546
fold 6 training in progress
fold 6 training completed, accuracy =  0.9318181818181818
fold 7 training in progress
fold 7 training completed, accuracy =  0.9090909090909091
fold 8 training in progress
fold 8 training completed, accuracy =  0.9767441860465116
fold 9 training in progress
fold 9 training completed, accuracy =  0.9761904761904762
fold 10 training in progress
fold 10 training completed, accuracy =  0.9761904761904762
```

[ ]:
```
accuarcy1_8, precision1_8, recall1_8, fscore_house1_8=␣
  ↪meanevaluation(listoflistofoutputs1_8,1)
print("First House Data Neural Network with " + str(hiddenlayerparemeter) + "␣
  ↪hidden layers and " + str(epoch) + " epochs")
print("accuracy:", acc1_8)
print("fscore:", fscore_house1_8)
plt.plot(range(epoch+1), listofjlist1_8[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("First House Data Neural Network with " + str(hiddenlayerparemeter) +␣
  ↪" hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
First House Data Neural Network with [16, 16, 16] hidden layers and 1500 epochs
accuracy: 0.9565488774791101
fscore: 0.9442298869694508
```



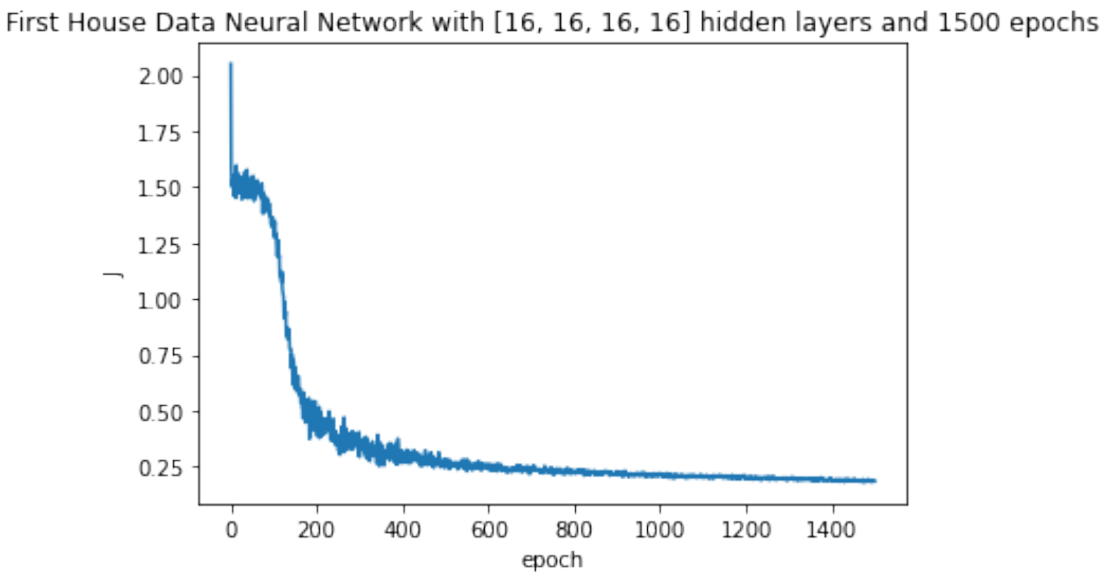
[ ]:
```
hiddenlayerparemeter = [16,16,16,16]
epoch = 1500
listoflistofoutputs1_9, acc1_9, listofjlist1_9 =␣
  ↪kfoldcrossvalidneuralnetwork(housedata, housecategory, hiddenlayerparemeter,␣
  ↪k = 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
  ↪0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9545454545454546
fold 2 training in progress
fold 2 training completed, accuracy =  0.9090909090909091
```

```
fold 3 training in progress
fold 3 training completed, accuracy =  1.0
fold 4 training in progress
fold 4 training completed, accuracy =  0.9545454545454546
fold 5 training in progress
fold 5 training completed, accuracy =  0.9545454545454546
fold 6 training in progress
fold 6 training completed, accuracy =  0.9318181818181818
fold 7 training in progress
fold 7 training completed, accuracy =  0.9318181818181818
fold 8 training in progress
fold 8 training completed, accuracy =  1.0
fold 9 training in progress
fold 9 training completed, accuracy =  0.9523809523809523
fold 10 training in progress
fold 10 training completed, accuracy =  0.9523809523809523
```

[ ]:
```
accuracy1_9, precision1_9, recall1_9, fscore_house1_9=
 ↪meanevaluation(listoflistofoutputs1_9,1)
print("First House Data Neural Network with " + str(hiddenlayerparemeter) + "
 ↪hidden layers and " + str(epoch) + " epochs")
print("accuracy:", acc1_9)
print("fscore:", fscore_house1_9)
plt.plot(range(epoch+1), listofjlist1_9[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("First House Data Neural Network with " + str(hiddenlayerparemeter) +
 ↪" hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
First House Data Neural Network with [16, 16, 16, 16] hidden layers and 1500
epochs
accuracy: 0.9541125541125541
fscore: 0.9410304727951788
```

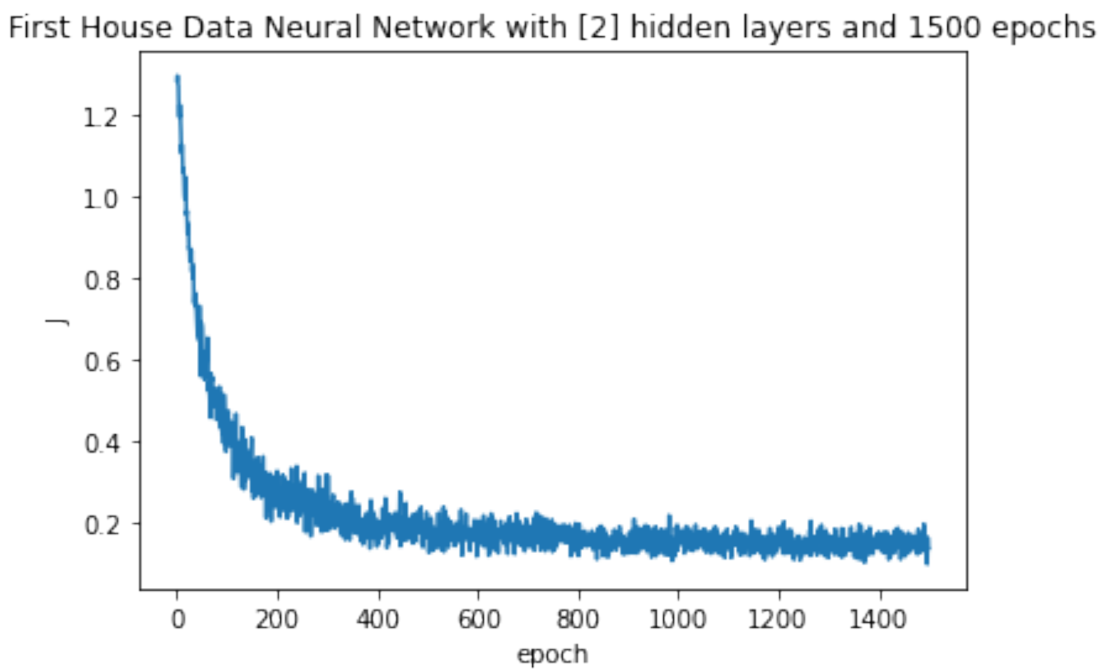



```
[ ]: hiddenlayerparemeter = [2]
epoch = 1500
listoflistofoutputs1_10, acc1_10, listofjlist1_10 =␣
 ↪kfoldcrossvalidneuralnetwork(housedata, housecategory, hiddenlayerparemeter,␣
 ↪k = 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
 ↪0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9545454545454546
fold 2 training in progress
fold 2 training completed, accuracy =  1.0
fold 3 training in progress
fold 3 training completed, accuracy =  1.0
fold 4 training in progress
fold 4 training completed, accuracy =  0.9545454545454546
fold 5 training in progress
fold 5 training completed, accuracy =  0.9545454545454546
fold 6 training in progress
fold 6 training completed, accuracy =  0.9545454545454546
fold 7 training in progress
fold 7 training completed, accuracy =  0.9772727272727273
fold 8 training in progress
fold 8 training completed, accuracy =  0.9069767441860465
fold 9 training in progress
fold 9 training completed, accuracy =  0.9523809523809523
fold 10 training in progress
fold 10 training completed, accuracy =  0.9285714285714286
```

```
[ ]: accuarcy1_10, precision1_10, recall1_10, fscore_house1_10=␣
      ↪meanevaluation(listoflistofoutputs1_10,1)
     print("First House Data Neural Network with " + str(hiddenlayerparemeter) + "␣
      ↪hidden layers and " + str(epoch) + " epochs")
     print("accuracy:", acc1_10)
     print("fscore:", fscore_house1_10)
     plt.plot(range(epoch+1), listofjlist1_10[1])
     plt.xlabel("epoch")
     plt.ylabel("J")
     plt.title("First House Data Neural Network with " + str(hiddenlayerparemeter) +␣
      ↪" hidden layers and " + str(epoch) + " epochs")
     plt.show()
```

```
First House Data Neural Network with [2] hidden layers and 1500 epochs
accuracy: 0.9583383670592973
fscore: 0.9472742127153891
```



House Data | **Hidden Layers** | **(2)** | **(4)** | **(4,4)** | **(4,4,4)** | **(4,4,4,4)** | **(8)** | **(8,8)** | **(16,16)** | **(16,16,16)** | **(16,16,16,16)** | | :— | :— | :— | :— | :— | :— | :— | :— | :— | :— | :— | | **Epoch** | 1500 | 1500 | 1500 | 1000 | 1500 | 1500 | 1500 | 1500 | 1500 | 1500 | | **Accuracy** | 0.9583 | 0.9611| 0.9632 | 0.9520 | 0.7752 | 0.9610 | 0.9545 | 0.9587 | 0.9565 | 0.9541 | | **F-Score** | 0.9473| 0.9491 | 0.9513 | 0.9386 | 0.4598 | 0.9478 | 0.9424 | 0.9473 | 0.9442 | 0.9410 |

the learning rate of them are 0.09 and 0.1. (The only two 0.09 are the (16,16) and (16,16,16))

Analysis of House Data

House Data

Hidden Layers	**(2)**	**(4)**	**(4,4)**	**(4,4,4)**	**(4,4,4,4)**	**(8)**	**(8,8)**	**(16,16)**	**(16,16,16)**	**(16,16,16,16)**
Epoch	1500	1500	1500	1000	1500	1500	1500	1500	1500	1500
Accuracy	0.9583	0.9611	0.9632	0.9520	0.7752	0.9610	0.9545	0.9587	0.9565	0.9541
F-Score	0.9473	0.9491	0.9513	0.9386	0.4598	0.9478	0.9424	0.9473	0.9442	0.9410

Discuss (on a high level) what contributed the most to improving performance: changing the regularization parameter; adding more layers; having deeper networks with many layers but few neurons per layer? designing networks with few layers but many neurons per layer? Discuss any patterns that you may have encountered. Also, discuss whether there is a point where constructing and training more "sophisticated"/complex networks—i.e., larger networks—no longer improves performance (or worsens performance).

Based on the analyses above, discuss which neural network architecture you would select if you had to deploy such a classifier in real life. Explain your reasoning.

ANSWER - House Vote: From this House Voting Data, most of them Have an accuracy around 0.9, except for the hidden layer [4,4,4,4] one. So that actually means the for this certain data, the increase in layer doesn't improve the accuracy, so this would means, the classification would be boundary could be easily with few neurons.

If we check on the table, we could find out that the best performance is from model with 2 hidden layers [4,4], with accuracy 0.9632 and F-score 0.9513, which also tell us, for this dataset, increasing the # of neurons per layer also doesn't increase the improvements a lot.

Indeed my assumption is that the data might even work well with logistic regression (that is, empty hidden layer[], directly calulate the output from input).

As for the graph, there are some interesting points. We can see that when the # of layer are at 1 or 2, the j graph is more 'smooth', that means it converge to a final weight without reaching and get stuck at some local minimum or so. (Heuristic might solve it but I didn't implement that)

Take a look of [16,16,16,16],[4,4,4,4],[16,16,16], we can see they've all been get to some local minimum stage, while others didn't or they escape from the local minimum very quick.

In real life, I will just use the [4,4] hidden layer, but maybe I will train using slightly smaller learning rate and more epochs so it converges better.

1.1.2 WINE Data

Trained with MiniBatch, vectorized neural network, divided to 3 group, (batchsize around 60)

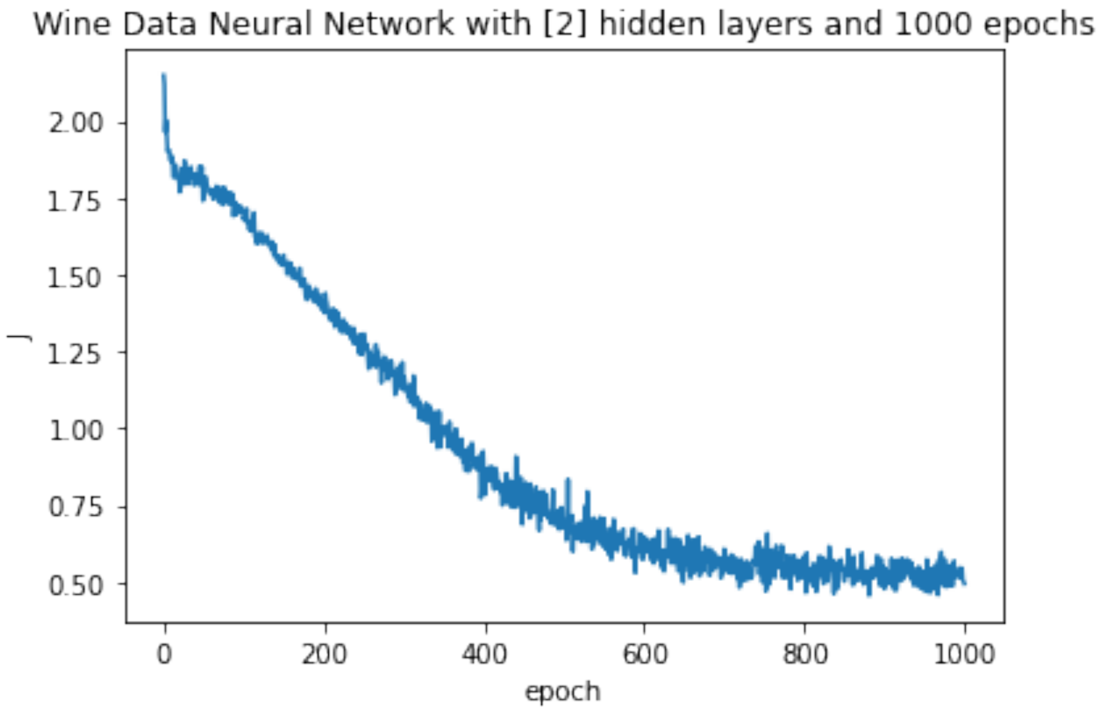
[ ]:
```
hiddenlayerparemeter = [2]
epoch = 1000
listoflistofoutputs2_1, acc2_1, listofjlist2_1 =␣
↪kfoldcrossvalidneuralnetwork(winedata, winecategory, hiddenlayerparemeter, k␣
↪= 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
↪0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  1.0
fold 2 training in progress
fold 2 training completed, accuracy =  1.0
fold 3 training in progress
fold 3 training completed, accuracy =  1.0
fold 4 training in progress
```

```
fold 4 training completed, accuracy =  0.9444444444444444
fold 5 training in progress
fold 5 training completed, accuracy =  1.0
fold 6 training in progress
fold 6 training completed, accuracy =  0.9444444444444444
fold 7 training in progress
fold 7 training completed, accuracy =  1.0
fold 8 training in progress
fold 8 training completed, accuracy =  1.0
fold 9 training in progress
fold 9 training completed, accuracy =  0.9411764705882353
fold 10 training in progress
fold 10 training completed, accuracy =  1.0
```

[ ]:
```
accuracy2_1, precision2_1, recall2_1, fscore_wine2_1=␣
  ↪meanevaluation(listoflistofoutputs2_1,1)
print("Wine Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
  ↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc2_1)))
print("fscore:", float("{0:.4f}". format(fscore_wine2_1)))
plt.plot(range(epoch+1), listofjlist2_1[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Wine Data Neural Network with " + str(hiddenlayerparemeter) + "␣
  ↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Wine Data Neural Network with [2] hidden layers and 1000 epochs
accuracy: 0.983
fscore: 0.9818
```



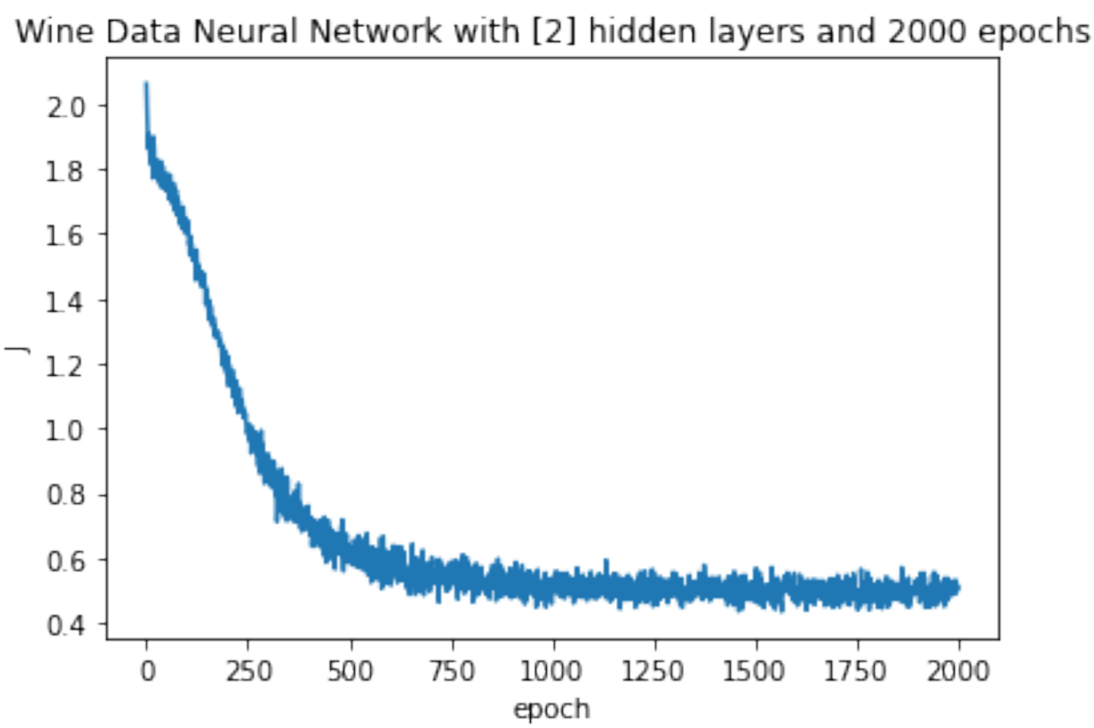
[ ]:
```
hiddenlayerparemeter = [2]
epoch = 2000
listoflistofoutputs2_2, acc2_2, listofjlist2_2 = kfoldcrossvalidneuralnetwork(winedata, winecategory, hiddenlayerparemeter, k = 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9473684210526315
fold 2 training in progress
fold 2 training completed, accuracy =  1.0
fold 3 training in progress
fold 3 training completed, accuracy =  1.0
fold 4 training in progress
fold 4 training completed, accuracy =  0.9444444444444444
fold 5 training in progress
fold 5 training completed, accuracy =  0.9444444444444444
fold 6 training in progress
fold 6 training completed, accuracy =  1.0
fold 7 training in progress
fold 7 training completed, accuracy =  1.0
fold 8 training in progress
fold 8 training completed, accuracy =  1.0
fold 9 training in progress
```

```
fold 9 training completed, accuracy =  1.0
fold 10 training in progress
fold 10 training completed, accuracy =  1.0
```

[ ]:
```
accuracy2_2, precision2_2, recall2_2, fscore_wine2_2=␣
 ↪meanevaluation(listoflistofoutputs2_2,1)
print("Wine Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
 ↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc2_2)))
print("fscore:", float("{0:.4f}". format(fscore_wine2_2)))
plt.plot(range(epoch+1), listofjlist2_2[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Wine Data Neural Network with " + str(hiddenlayerparemeter) + "␣
 ↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Wine Data Neural Network with [2] hidden layers and 2000 epochs
accuracy: 0.9836
fscore: 0.9818
```



[ ]:
```
hiddenlayerparemeter = [2]
epoch = 500
```

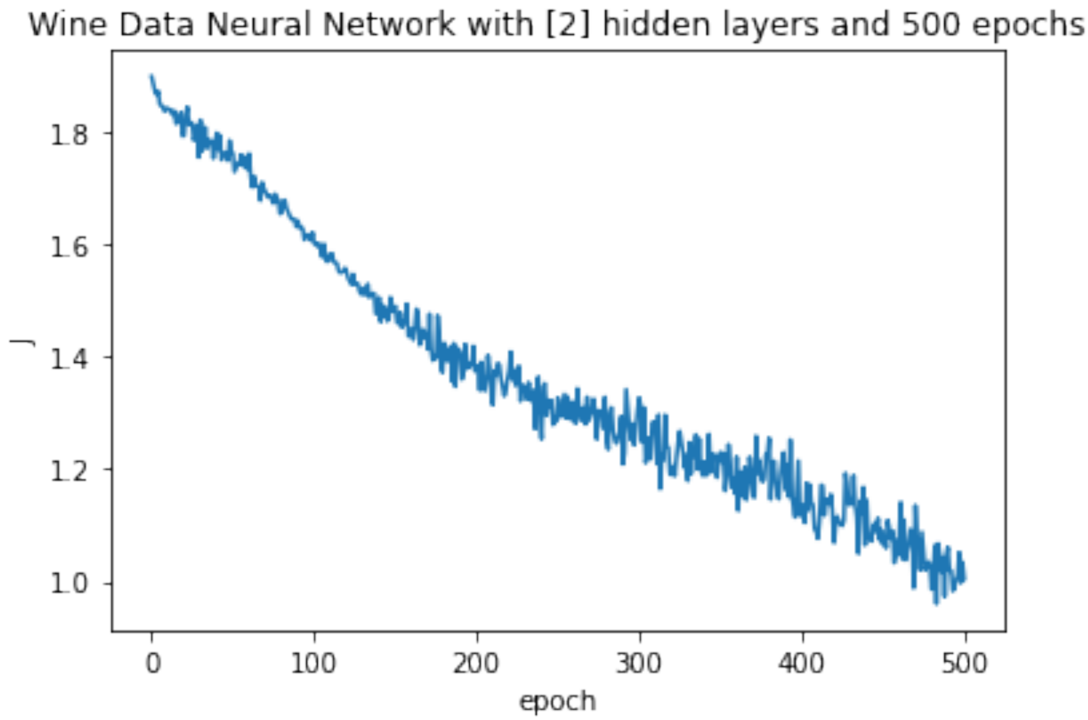
```
listoflistofoutputs2_3, acc2_3, listofjlist2_3 =␣
↪kfoldcrossvalidneuralnetwork(winedata, winecategory, hiddenlayerparemeter, k␣
↪= 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
↪0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9473684210526315
fold 2 training in progress
fold 2 training completed, accuracy =  0.8888888888888888
fold 3 training in progress
fold 3 training completed, accuracy =  0.9444444444444444
fold 4 training in progress
fold 4 training completed, accuracy =  1.0
fold 5 training in progress
fold 5 training completed, accuracy =  0.8888888888888888
fold 6 training in progress
fold 6 training completed, accuracy =  1.0
fold 7 training in progress
fold 7 training completed, accuracy =  1.0
fold 8 training in progress
fold 8 training completed, accuracy =  0.9444444444444444
fold 9 training in progress
fold 9 training completed, accuracy =  0.9411764705882353
fold 10 training in progress
fold 10 training completed, accuracy =  1.0
```

[ ]:

```
accuracy2_3, precision2_3, recall2_3, fscore_wine2_3=␣
↪meanevaluation(listoflistofoutputs2_3,1)
print("Wine Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc2_3)))
print("fscore:", float("{0:.4f}". format(fscore_wine2_3)))
plt.plot(range(epoch+1), listofjlist2_3[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Wine Data Neural Network with " + str(hiddenlayerparemeter) + "␣
↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Wine Data Neural Network with [2] hidden layers and 500 epochs
accuracy: 0.9555
fscore: 0.939
```



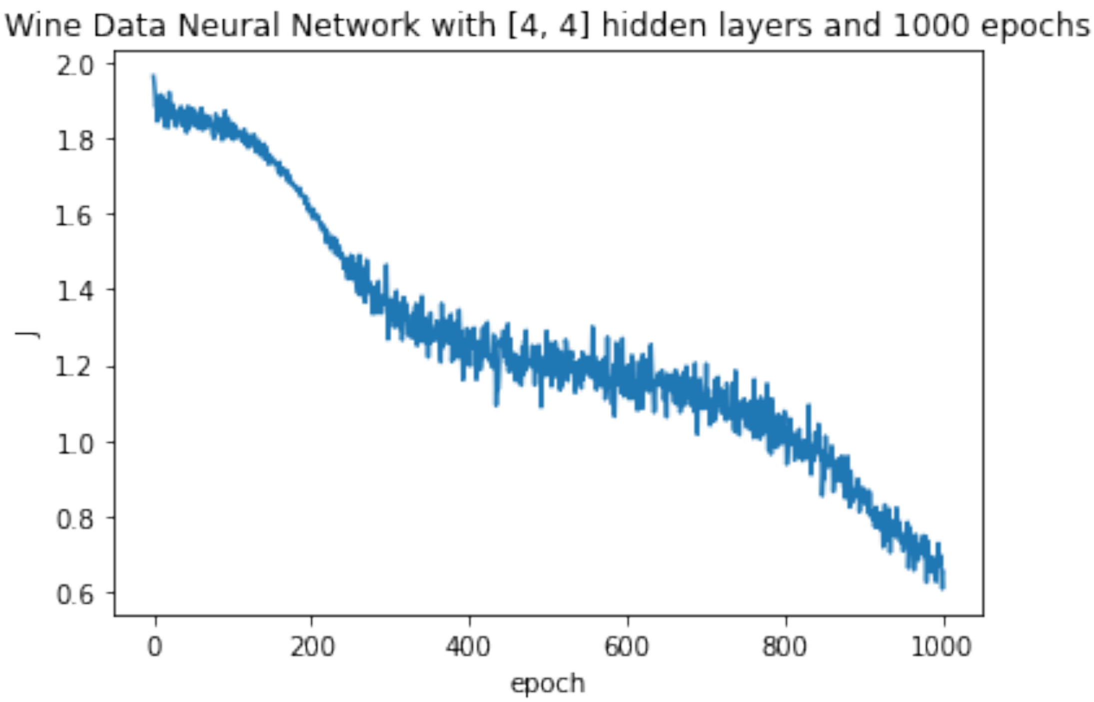
[ ]:
```
hiddenlayerparemeter = [4,4]
epoch = 1000
listoflistofoutputs2_4, acc2_4, listofjlist2_4 =␣
↪kfoldcrossvalidneuralnetwork(winedata, winecategory, hiddenlayerparemeter, k␣
↪= 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
↪0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  1.0
fold 2 training in progress
fold 2 training completed, accuracy =  1.0
fold 3 training in progress
fold 3 training completed, accuracy =  0.8888888888888888
fold 4 training in progress
fold 4 training completed, accuracy =  0.9444444444444444
fold 5 training in progress
fold 5 training completed, accuracy =  0.7222222222222222
fold 6 training in progress
fold 6 training completed, accuracy =  1.0
fold 7 training in progress
fold 7 training completed, accuracy =  0.9444444444444444
fold 8 training in progress
fold 8 training completed, accuracy =  1.0
fold 9 training in progress
```

```
fold 9 training completed, accuracy =  1.0
fold 10 training in progress
fold 10 training completed, accuracy =  1.0
```

[ ]:
```
accuracy2_4, precision2_4, recall2_4, fscore_wine2_4=␣
↪meanevaluation(listoflistofoutputs2_4,1)
print("Wine Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc2_4)))
print("fscore:", float("{0:.4f}". format(fscore_wine2_4)))
plt.plot(range(epoch+1), listofjlist2_4[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Wine Data Neural Network with " + str(hiddenlayerparemeter) + "␣
↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Wine Data Neural Network with [4, 4] hidden layers and 1000 epochs
accuracy: 0.95
fscore: 0.979
```



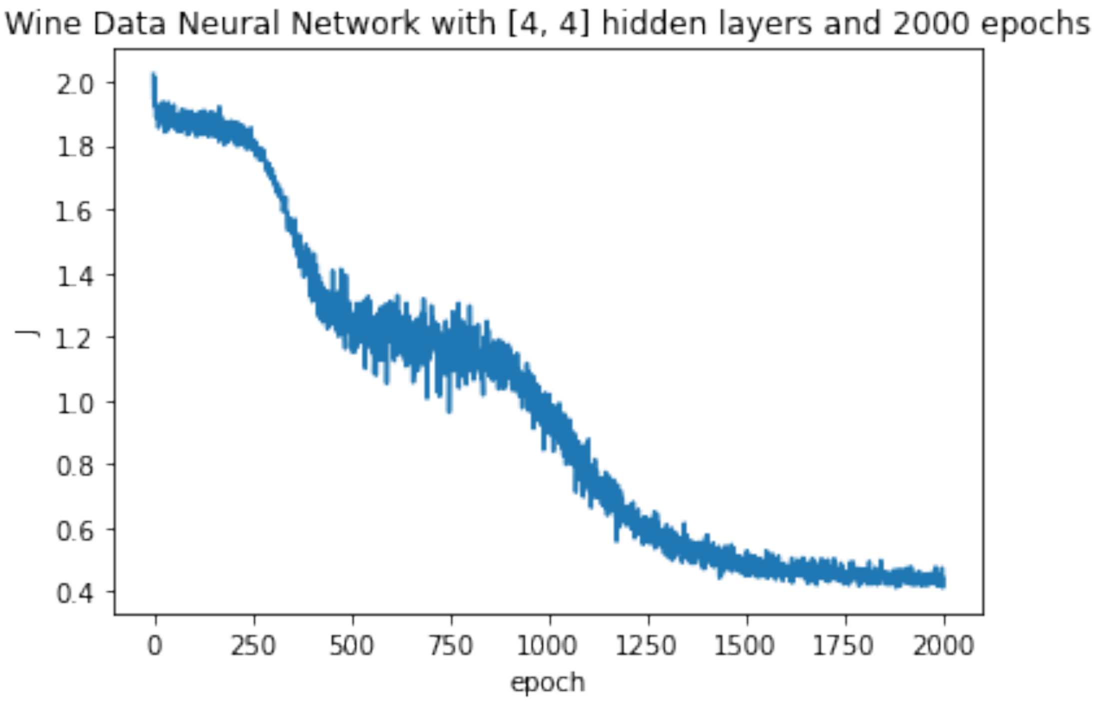
[ ]:
```
hiddenlayerparemeter = [4,4]
epoch = 2000
```

```
listoflistofoutputs2_5, acc2_5, listofjlist2_5 =␣
↪kfoldcrossvalidneuralnetwork(winedata, winecategory, hiddenlayerparemeter, k␣
↪= 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
↪0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  1.0
fold 2 training in progress
fold 2 training completed, accuracy =  0.9444444444444444
fold 3 training in progress
fold 3 training completed, accuracy =  1.0
fold 4 training in progress
fold 4 training completed, accuracy =  1.0
fold 5 training in progress
fold 5 training completed, accuracy =  1.0
fold 6 training in progress
fold 6 training completed, accuracy =  0.9444444444444444
fold 7 training in progress
fold 7 training completed, accuracy =  0.9444444444444444
fold 8 training in progress
fold 8 training completed, accuracy =  1.0
fold 9 training in progress
fold 9 training completed, accuracy =  1.0
fold 10 training in progress
fold 10 training completed, accuracy =  1.0
```

[ ]:
```
accuracy2_5, precision2_5, recall2_5, fscore_wine2_5=␣
↪meanevaluation(listoflistofoutputs2_5,1)
print("Wine Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc2_5)))
print("fscore:", float("{0:.4f}". format(fscore_wine2_5)))
plt.plot(range(epoch+1), listofjlist2_5[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Wine Data Neural Network with " + str(hiddenlayerparemeter) + "␣
↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Wine Data Neural Network with [4, 4] hidden layers and 2000 epochs
accuracy: 0.9833
fscore: 0.9779
```



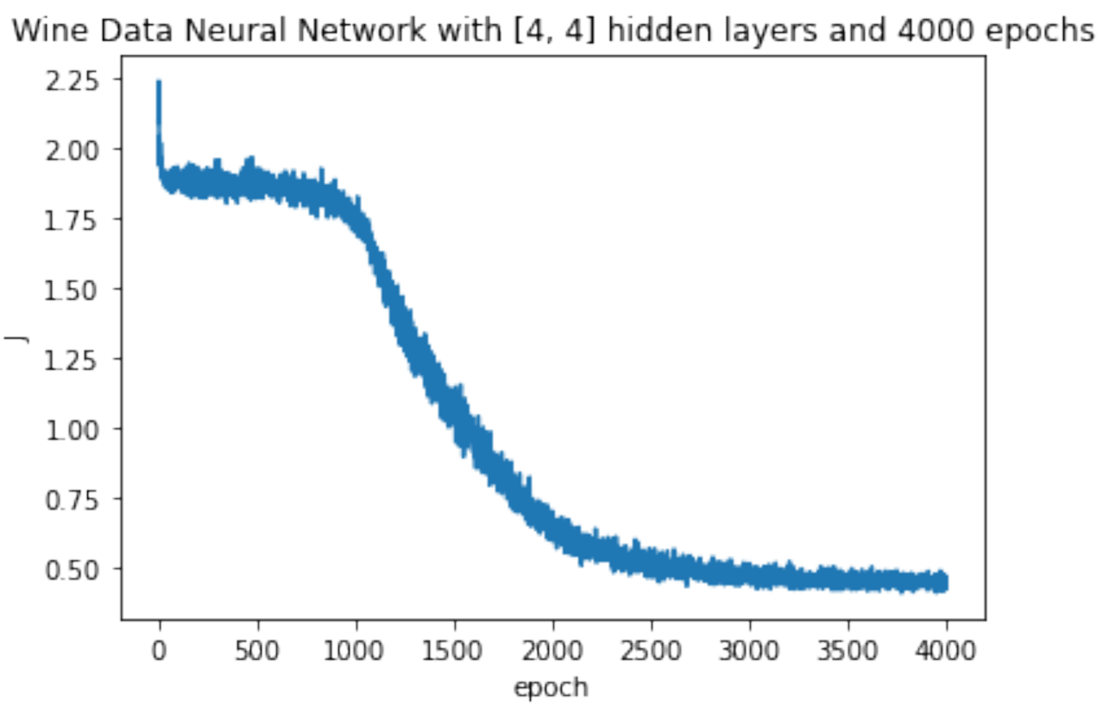
[ ]:
```
hiddenlayerparemeter = [4,4]
epoch = 4000
listoflistofoutputs2_6, acc2_6, listofjlist2_6 =␣
↪kfoldcrossvalidneuralnetwork(winedata, winecategory, hiddenlayerparemeter, k␣
↪= 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.05, epsilon_0 = 0.
↪0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  1.0
fold 2 training in progress
fold 2 training completed, accuracy =  1.0
fold 3 training in progress
fold 3 training completed, accuracy =  0.9444444444444444
fold 4 training in progress
fold 4 training completed, accuracy =  1.0
fold 5 training in progress
fold 5 training completed, accuracy =  1.0
fold 6 training in progress
fold 6 training completed, accuracy =  1.0
fold 7 training in progress
fold 7 training completed, accuracy =  1.0
fold 8 training in progress
fold 8 training completed, accuracy =  1.0
fold 9 training in progress
```

```
fold 9 training completed, accuracy =  0.9411764705882353
fold 10 training in progress
fold 10 training completed, accuracy =  1.0
```

[ ]:
```
accuracy2_6, precision2_6, recall2_6, fscore_wine2_6=␣
 ↪meanevaluation(listoflistofoutputs2_6,1)
print("Wine Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
 ↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc2_6)))
print("fscore:", float("{0:.4f}". format(fscore_wine2_6)))
plt.plot(range(epoch+1), listofjlist2_6[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Wine Data Neural Network with " + str(hiddenlayerparemeter) + "␣
 ↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Wine Data Neural Network with [4, 4] hidden layers and 4000 epochs
accuracy: 0.9886
fscore: 0.9856
```



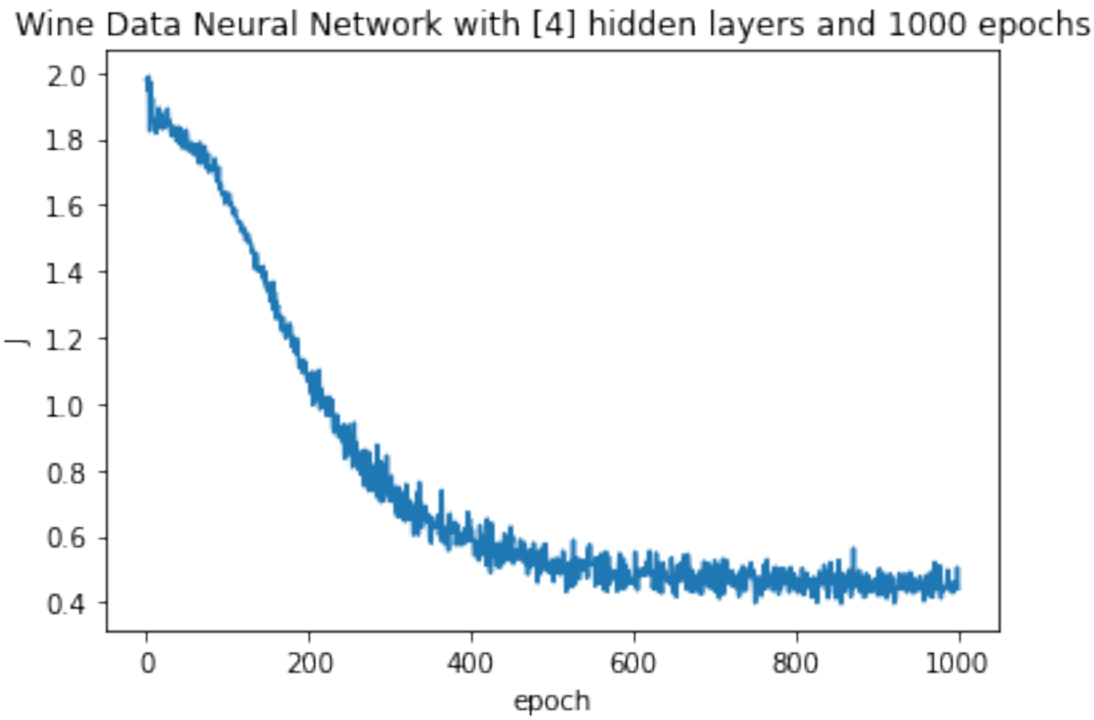
[ ]:
```
hiddenlayerparemeter = [4]
epoch = 1000
```

```
listoflistofoutputs2_7, acc2_7, listofjlist2_7 =␣
↪kfoldcrossvalidneuralnetwork(winedata, winecategory, hiddenlayerparemeter, k␣
↪= 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
↪0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  1.0
fold 2 training in progress
fold 2 training completed, accuracy =  1.0
fold 3 training in progress
fold 3 training completed, accuracy =  1.0
fold 4 training in progress
fold 4 training completed, accuracy =  0.9444444444444444
fold 5 training in progress
fold 5 training completed, accuracy =  1.0
fold 6 training in progress
fold 6 training completed, accuracy =  1.0
fold 7 training in progress
fold 7 training completed, accuracy =  1.0
fold 8 training in progress
fold 8 training completed, accuracy =  1.0
fold 9 training in progress
fold 9 training completed, accuracy =  0.9411764705882353
fold 10 training in progress
fold 10 training completed, accuracy =  0.9375
```

[ ]:
```
accuracy2_7, precision2_7, recall2_7, fscore_wine2_7=␣
↪meanevaluation(listoflistofoutputs2_7,1)
print("Wine Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc2_7)))
print("fscore:", float("{0:.4f}". format(fscore_wine2_7)))
plt.plot(range(epoch+1), listofjlist2_7[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Wine Data Neural Network with " + str(hiddenlayerparemeter) + "␣
↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Wine Data Neural Network with [4] hidden layers and 1000 epochs
accuracy: 0.9823
fscore: 0.9779
```



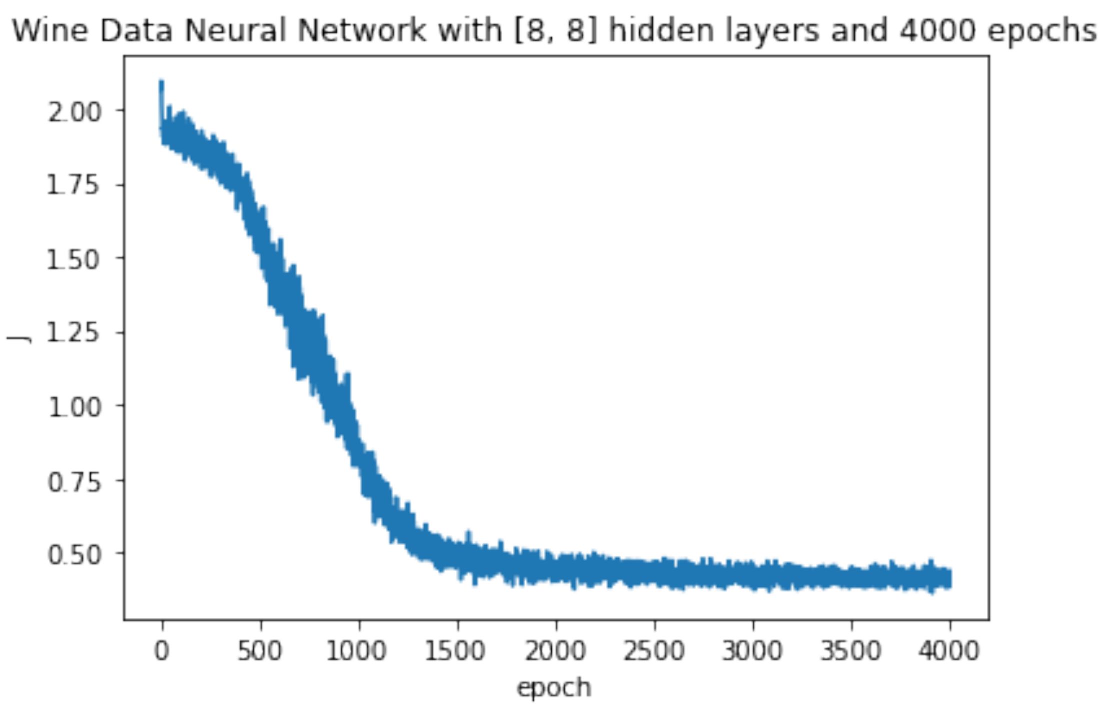
[ ]:
```
hiddenlayerparemeter = [8,8]
epoch = 4000
listoflistofoutputs2_8, acc2_8, listofjlist2_8 =␣
↪kfoldcrossvalidneuralnetwork(winedata, winecategory, hiddenlayerparemeter, k␣
↪= 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.05, epsilon_0 = 0.
↪0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9473684210526315
fold 2 training in progress
fold 2 training completed, accuracy =  1.0
fold 3 training in progress
fold 3 training completed, accuracy =  0.9444444444444444
fold 4 training in progress
fold 4 training completed, accuracy =  0.9444444444444444
fold 5 training in progress
fold 5 training completed, accuracy =  1.0
fold 6 training in progress
fold 6 training completed, accuracy =  1.0
fold 7 training in progress
fold 7 training completed, accuracy =  1.0
fold 8 training in progress
fold 8 training completed, accuracy =  1.0
fold 9 training in progress
```

```
fold 9 training completed, accuracy =  1.0
fold 10 training in progress
fold 10 training completed, accuracy =  1.0
```

[ ]:
```python
accuracy2_8, precision2_8, recall2_8, fscore_wine2_8=␣
↪meanevaluation(listoflistofoutputs2_8,1)
print("Wine Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc2_8)))
print("fscore:", float("{0:.4f}". format(fscore_wine2_8)))
plt.plot(range(epoch+1), listofjlist2_8[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Wine Data Neural Network with " + str(hiddenlayerparemeter) + "␣
↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Wine Data Neural Network with [8, 8] hidden layers and 4000 epochs
accuracy: 0.9836
fscore: 0.979
```



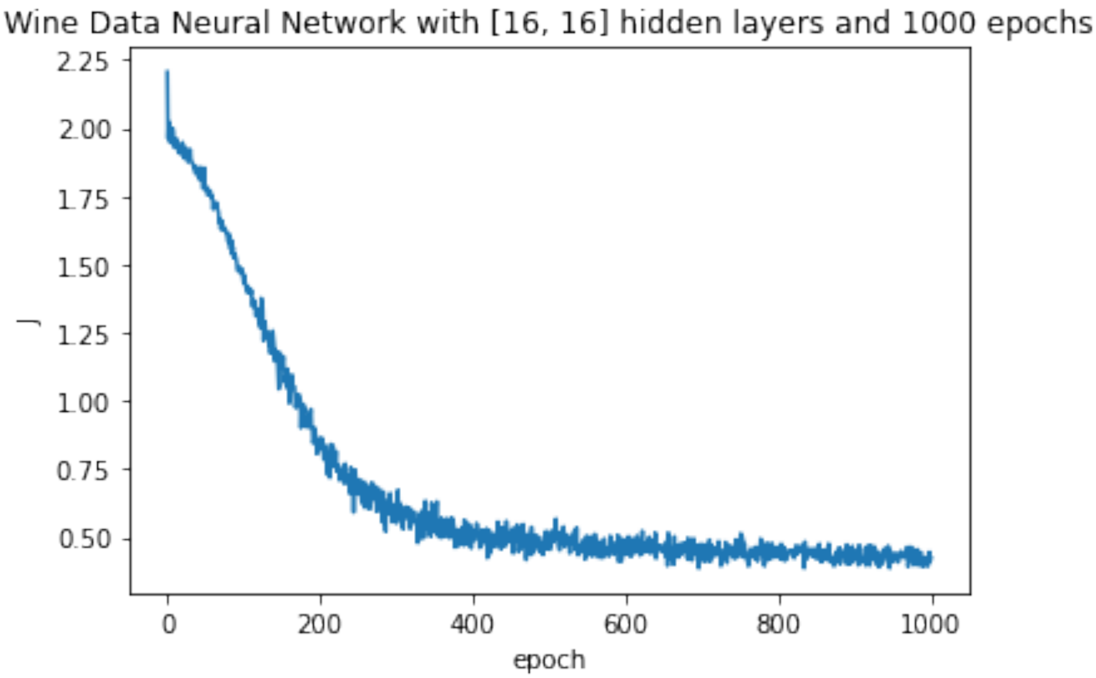
[ ]:
```python
hiddenlayerparemeter = [16,16]
epoch = 1000
```

```
listoflistofoutputs2_9, acc2_9, listofjlist2_9 =␣
↪kfoldcrossvalidneuralnetwork(winedata, winecategory, hiddenlayerparemeter, k␣
↪= 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
↪0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9473684210526315
fold 2 training in progress
fold 2 training completed, accuracy =  1.0
fold 3 training in progress
fold 3 training completed, accuracy =  0.9444444444444444
fold 4 training in progress
fold 4 training completed, accuracy =  1.0
fold 5 training in progress
fold 5 training completed, accuracy =  1.0
fold 6 training in progress
fold 6 training completed, accuracy =  0.8888888888888888
fold 7 training in progress
fold 7 training completed, accuracy =  1.0
fold 8 training in progress
fold 8 training completed, accuracy =  1.0
fold 9 training in progress
fold 9 training completed, accuracy =  1.0
fold 10 training in progress
fold 10 training completed, accuracy =  1.0
```

[ ]:
```
accuracy2_9, precision2_9, recall2_9, fscore_wine2_9=␣
↪meanevaluation(listoflistofoutputs2_9,1)
print("Wine Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc2_9)))
print("fscore:", float("{0:.4f}". format(fscore_wine2_9)))
plt.plot(range(epoch+1), listofjlist2_9[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Wine Data Neural Network with " + str(hiddenlayerparemeter) + "␣
↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Wine Data Neural Network with [16, 16] hidden layers and 1000 epochs
accuracy: 0.9781
fscore: 0.9698
```



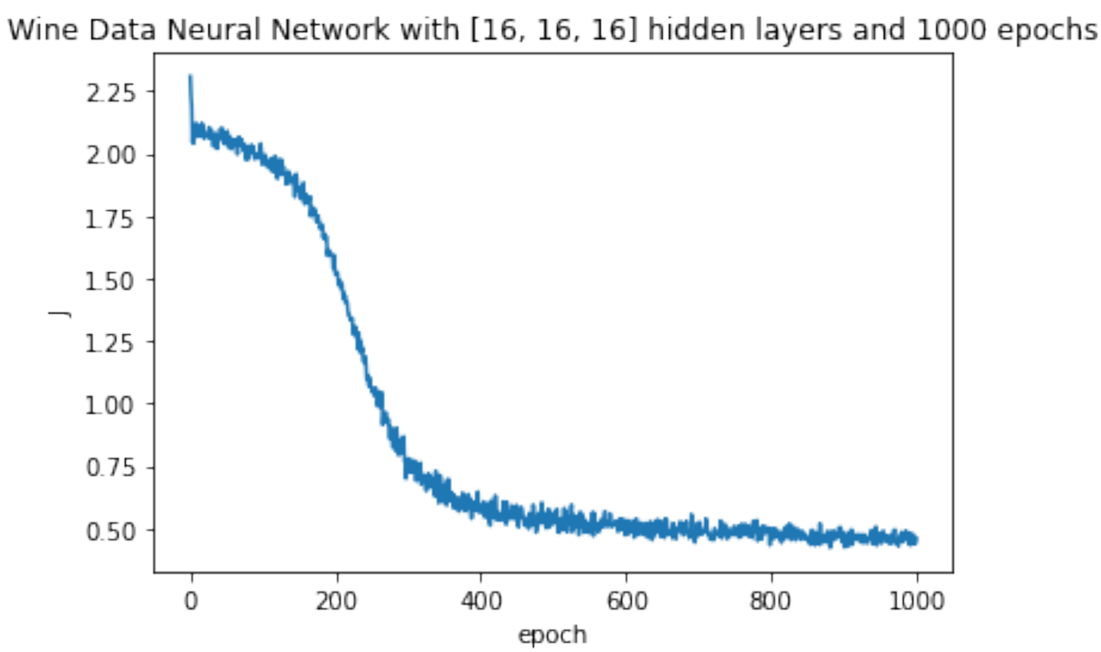
[ ]:
```
hiddenlayerparemeter = [16,16,16]
epoch = 1000
listoflistofoutputs2_10, acc2_10, listofjlist2_10 = kfoldcrossvalidneuralnetwork(winedata, winecategory, hiddenlayerparemeter, k = 10, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  1.0
fold 2 training in progress
fold 2 training completed, accuracy =  1.0
fold 3 training in progress
fold 3 training completed, accuracy =  1.0
fold 4 training in progress
fold 4 training completed, accuracy =  1.0
fold 5 training in progress
fold 5 training completed, accuracy =  0.8888888888888888
fold 6 training in progress
fold 6 training completed, accuracy =  1.0
fold 7 training in progress
fold 7 training completed, accuracy =  1.0
fold 8 training in progress
fold 8 training completed, accuracy =  0.8888888888888888
fold 9 training in progress
fold 9 training completed, accuracy =  1.0
```

```
fold 10 training in progress
fold 10 training completed, accuracy =  1.0
```

[ ]:
```
accuracy2_10, precision2_10, recall2_10, fscore_wine2_10=␣
  ↪meanevaluation(listoflistofoutputs2_10,1)
print("Wine Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
  ↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc2_10)))
print("fscore:", float("{0:.4f}". format(fscore_wine2_10)))
plt.plot(range(epoch+1), listofjlist2_10[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Wine Data Neural Network with " + str(hiddenlayerparemeter) + "␣
  ↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Wine Data Neural Network with [16, 16, 16] hidden layers and 1000 epochs
accuracy: 0.9778
fscore: 0.9667
```



WINE Data | **Hidden Layers** | **(2)** | **(2)** | **(2)** | **(4)** | **(4,4)** | **(4,4)** | **(4,4) LR=0.05** | **(8,8) LR=0.05** | **(16,16)** | **(16,16,16)** | | | :— | :— | :— | :— | :— | :— | :— | :— | :— | :— | :— | | | **Epoch** | 1000 | 2000 | 500 | 1000 | 1000 | 2000 | 4000 | 4000 | 1000 | 1000 | | | **Accuracy** | 0.9830 | 0.9836| 0.9555 | 0.9823 | 0.9500 | 0.9833 | 0.9886 | 0.9836 | 0.9781 | 0.9778 | | | **F-Score** | 0.9818| 0.9818 | 0.9390 | 0.9779 | 0.9790 | 0.9779 | 0.9856 | 0.9790 | 0.9698 | 0.9667 |

the learning rate of them are 0.05 and 0.1. (The two 0.05 have the label 'LR=0.5')

WINE Data

Hidden Layers	(2)	(2)	(2)	(4)	(4,4)	(4,4)	(4,4) LR=0.05	(8,8) LR=0.05	(16,16)	(16,16,16)
Epoch	1000	2000	500	1000	1000	2000	4000	4000	1000	1000
Accuracy	0.9830	0.9836	0.9555	0.9823	0.9500	0.9833	0.9886	0.9836	0.9781	0.9778
F-Score	0.9818	0.9818	0.9390	0.9779	0.9790	0.9779	0.9856	0.9790	0.9698	0.9667

Analysis of WINE Data

> Discuss (on a high level) what contributed the most to improving performance: changing the regularization parameter; adding more layers; having deeper networks with many layers but few neurons per layer? designing networks with few layers but many neurons per layer? Discuss any patterns that you may have encountered. Also, discuss whether there is a point where constructing and training more "sophisticated"/complex networks—i.e., larger networks—no longer improves performance (or worsens performance).
>
> Based on the analyses above, discuss which neural network architecture you would select if you had to deploy such a classifier in real life. Explain your reasoning.

ANSWER: The performance of the neural network for wine dataset are generally really good, with around 0.98 accuracy and f-score. For these group of tests, I am and trying to change the epoch, learning rate and layer data to see what are the contributions of them to the performance of the network.

For the first three (columns), I tested them with same single hidden layer [2] with different epochs. As we see from the value, with epoch = 500, the performance is around 0.95, and when epoch=1000, accuracy is around 0.983, and when epoch=2000, accuracy is about 0.9836, what demonstrate that with higher e, the j function converge to it's smallest value. As shown in the plot figures, we can see how j converge for all the [2] hidden layers neural network.

Then, I tried to change the # of layers to see what's the difference are, so, for [4] and [4,4], with both 1000 epochs, we can see that [4] have better performance than [4,4]. My reasoning is that with more layers and neurons, there will be more weights, so that would the model more epochs to converge to a minimum value, but we set both to 1000, so it still hasn't converged to a nice value yet.

Then I decided to modify the learning rate of the data, with modifyine the learning rate of [4,4] from 0.1 to 0.05, I doubled the epoch because intuitively, I think the learning rate will make the gradient descent process slower, so we need more epoch to converge. And it turns out that the performance get even better, accuracy improved from 0.983 to 0.989, and f-score improved from 0.978 to 0.986.

Then, I changed to [4,4] to [8,8], holding other parameter unchanged. And the performance didn't improve. So that shows us the max performance is around [4,4] with one or two layers.

In real world, I might just use the [4] with 1000 epochs, since the performance 0.982 is already good enough,

considering it take less time to train than the [4,4] 2000-epochs network.

1.2 EXTRA CREDITS

1.2.1 EXTRA I: vectorized form of backpropagation

I Implemented backpropagation using vectors.

1.2.2 EXTRA II: CANCER Data

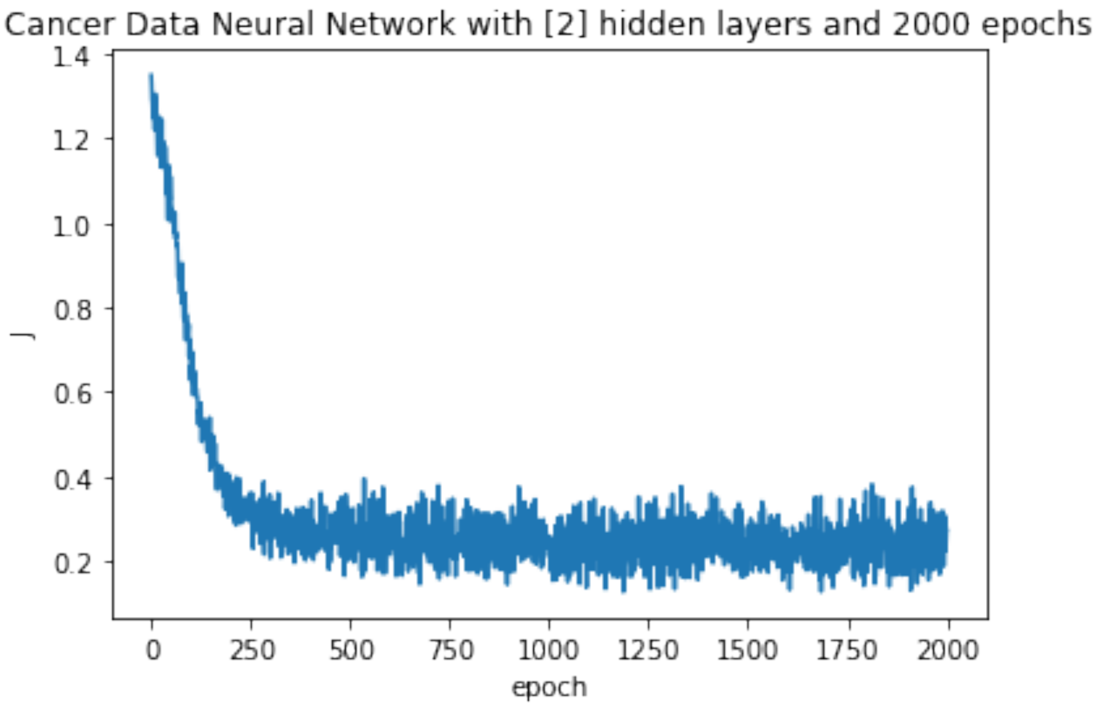
Trained with MiniBatch, vectorized neural network, divided to 5 group, (batchsize around 130-140)

[ ]:
```
hiddenlayerparemeter = [2]
epoch = 2000
listoflistofoutputs3_1, acc3_1, listofjlist3_1 =␣
 ↪kfoldcrossvalidneuralnetwork(cancerdata, cancercategory,␣
 ↪hiddenlayerparemeter, k = 10, minibatchk = 3, lambda_reg = 0.1,␣
 ↪learning_rate = 0.1, epsilon_0 = 0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  1.0
fold 2 training in progress
fold 2 training completed, accuracy =  0.9857142857142858
fold 3 training in progress
fold 3 training completed, accuracy =  0.9285714285714286
fold 4 training in progress
fold 4 training completed, accuracy =  0.9571428571428572
fold 5 training in progress
fold 5 training completed, accuracy =  0.9714285714285714
fold 6 training in progress
fold 6 training completed, accuracy =  0.9714285714285714
fold 7 training in progress
fold 7 training completed, accuracy =  0.9428571428571428
fold 8 training in progress
fold 8 training completed, accuracy =  0.9714285714285714
fold 9 training in progress
fold 9 training completed, accuracy =  0.927536231884058
fold 10 training in progress
fold 10 training completed, accuracy =  0.9710144927536232
```

[ ]:
```
accuracy3_1, precision3_1, recall3_1, fscore_cancer3_1=␣
 ↪meanevaluation(listoflistofoutputs3_1,1)
print("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
 ↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc3_1)))
print("fscore:", float("{0:.4f}". format(fscore_cancer3_1)))
plt.plot(range(epoch+1), listofjlist3_1[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + "␣
 ↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Cancer Data Neural Network with [2] hidden layers and 2000 epochs
accuracy: 0.9627
fscore: 0.9469
```



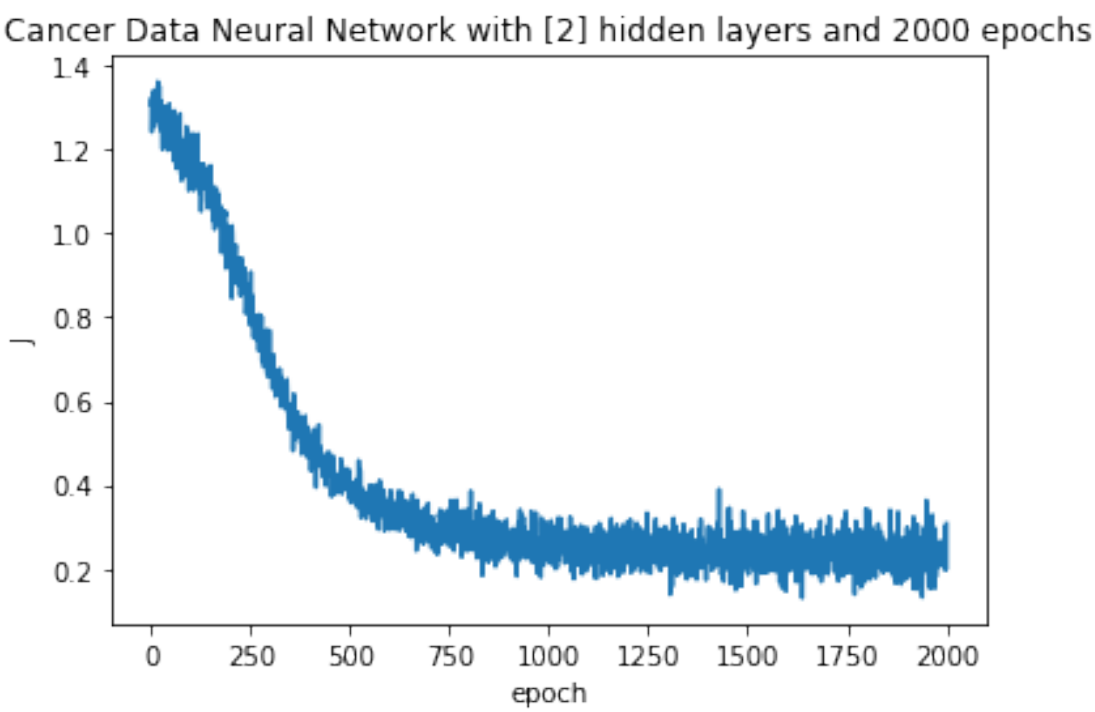
[ ]:
```
hiddenlayerparemeter = [2]
epoch = 2000
listoflistofoutputs3_2, acc3_2, listofjlist3_2 =␣
↪kfoldcrossvalidneuralnetwork(cancerdata, cancercategory,␣
↪hiddenlayerparemeter, k = 10, minibatchk = 3, lambda_reg = 0.1,␣
↪learning_rate = 0.05, epsilon_0 = 0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9436619718309859
fold 2 training in progress
fold 2 training completed, accuracy =  0.9571428571428572
fold 3 training in progress
fold 3 training completed, accuracy =  0.9857142857142858
fold 4 training in progress
fold 4 training completed, accuracy =  0.9857142857142858
fold 5 training in progress
fold 5 training completed, accuracy =  0.9857142857142858
fold 6 training in progress
fold 6 training completed, accuracy =  0.9571428571428572
fold 7 training in progress
fold 7 training completed, accuracy =  0.9
fold 8 training in progress
fold 8 training completed, accuracy =  0.9714285714285714
fold 9 training in progress
```

```
fold 9 training completed, accuracy =  0.9420289855072463
fold 10 training in progress
fold 10 training completed, accuracy =  0.9855072463768116
```

[ ]:
```
accuracy3_2, precision3_2, recall3_2, fscore_cancer3_2=␣
 ↪meanevaluation(listoflistofoutputs3_2,1)
print("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
 ↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc3_2)))
print("fscore:", float("{0:.4f}". format(fscore_cancer3_2)))
plt.plot(range(epoch+1), listofjlist3_2[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + "␣
 ↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Cancer Data Neural Network with [2] hidden layers and 2000 epochs
accuracy: 0.9614
fscore: 0.9447
```



[ ]:
```
hiddenlayerparemeter = [2]
epoch = 2000
```

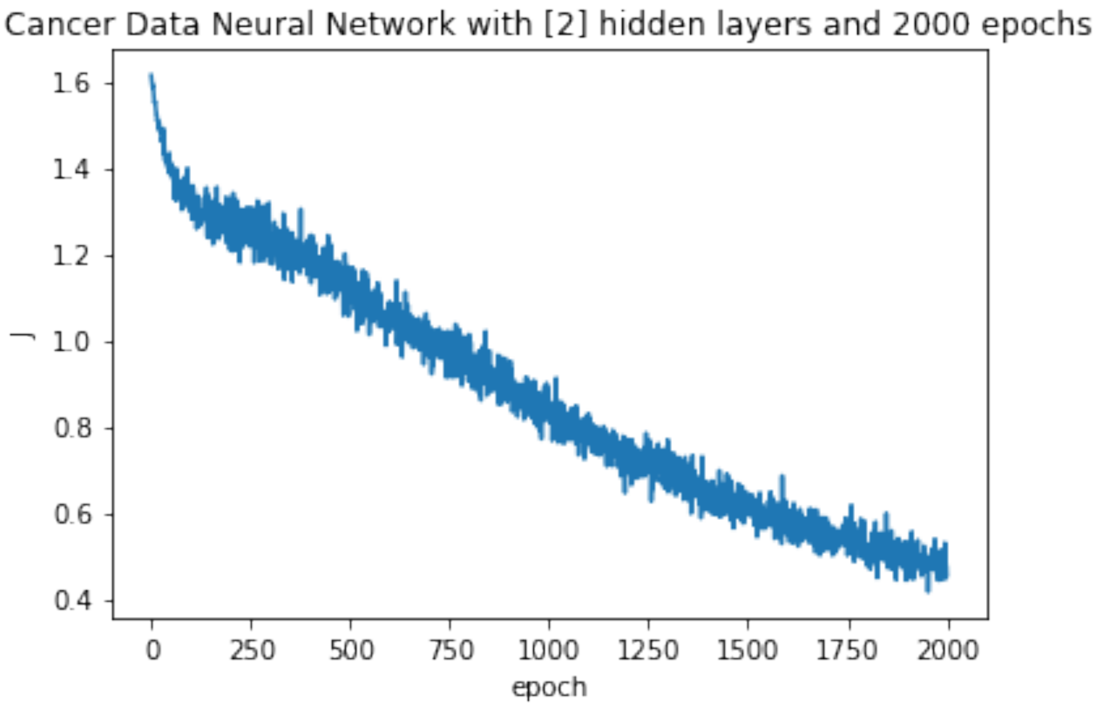
```
listoflistofoutputs3_3, acc3_3, listofjlist3_3 =␣
 ↪kfoldcrossvalidneuralnetwork(cancerdata, cancercategory,␣
 ↪hiddenlayerparemeter, k = 10, minibatchk = 3, lambda_reg = 0.1,␣
 ↪learning_rate = 0.01, epsilon_0 = 0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.971830985915493
fold 2 training in progress
fold 2 training completed, accuracy =  0.9714285714285714
fold 3 training in progress
fold 3 training completed, accuracy =  0.9571428571428572
fold 4 training in progress
fold 4 training completed, accuracy =  0.9857142857142858
fold 5 training in progress
fold 5 training completed, accuracy =  0.9571428571428572
fold 6 training in progress
fold 6 training completed, accuracy =  0.9571428571428572
fold 7 training in progress
fold 7 training completed, accuracy =  0.9285714285714286
fold 8 training in progress
fold 8 training completed, accuracy =  0.9285714285714286
fold 9 training in progress
fold 9 training completed, accuracy =  0.9420289855072463
fold 10 training in progress
fold 10 training completed, accuracy =  0.9710144927536232
```

[ ]:

```
accuracy3_3, precision3_3, recall3_3, fscore_cancer3_3=␣
 ↪meanevaluation(listoflistofoutputs3_3,1)
print("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
 ↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc3_3)))
print("fscore:", float("{0:.4f}". format(fscore_cancer3_3)))
plt.plot(range(epoch+1), listofjlist3_3[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + "␣
 ↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Cancer Data Neural Network with [2] hidden layers and 2000 epochs
accuracy: 0.9571
fscore: 0.9365
```



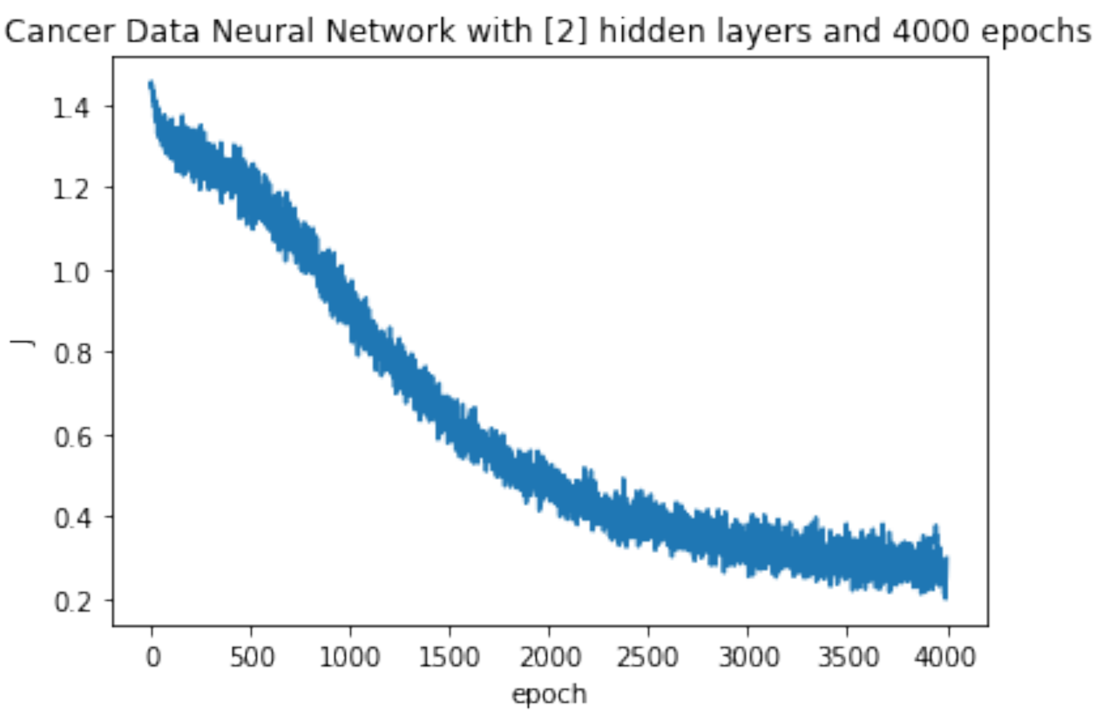

```
[ ]: hiddenlayerparemeter = [2]
epoch = 4000
listoflistofoutputs3_4, acc3_4, listofjlist3_4 =␣
 ↪kfoldcrossvalidneuralnetwork(cancerdata, cancercategory,␣
 ↪hiddenlayerparemeter, k = 10, minibatchk = 3, lambda_reg = 0.1,␣
 ↪learning_rate = 0.01, epsilon_0 = 0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.971830985915493
fold 2 training in progress
fold 2 training completed, accuracy =  0.9285714285714286
fold 3 training in progress
fold 3 training completed, accuracy =  0.9857142857142858
fold 4 training in progress
fold 4 training completed, accuracy =  0.9285714285714286
fold 5 training in progress
fold 5 training completed, accuracy =  0.9714285714285714
fold 6 training in progress
fold 6 training completed, accuracy =  0.9285714285714286
fold 7 training in progress
fold 7 training completed, accuracy =  0.9857142857142858
fold 8 training in progress
fold 8 training completed, accuracy =  0.9714285714285714
fold 9 training in progress
```

```
fold 9 training completed, accuracy =  0.9855072463768116
fold 10 training in progress
fold 10 training completed, accuracy =  0.9710144927536232
```

[ ]:
```
epoch = 4000
hiddenlayerparemeter = [2]
accuracy3_4, precision3_4, recall3_4, fscore_cancer3_4=␣
  ↪meanevaluation(listoflistofoutputs3_4,1)
print("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
  ↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc3_4)))
print("fscore:", float("{0:.4f}". format(fscore_cancer3_4)))
plt.plot(range(epoch+1), listofjlist3_4[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + "␣
  ↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Cancer Data Neural Network with [2] hidden layers and 4000 epochs
accuracy: 0.9628
fscore: 0.9462
```

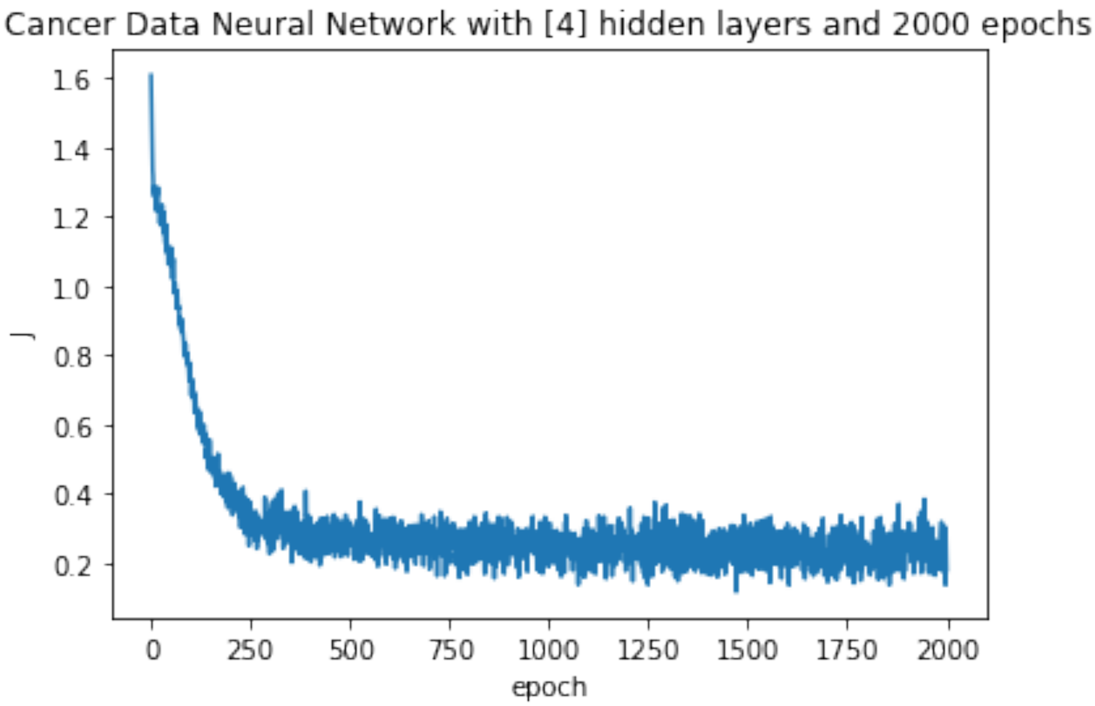


[ ]:
```
hiddenlayerparemeter = [4]
epoch = 2000
listoflistofoutputs3_5, acc3_5, listofjlist3_5 =␣
↪kfoldcrossvalidneuralnetwork(cancerdata, cancercategory,␣
↪hiddenlayerparemeter, k = 10, minibatchk = 3, lambda_reg = 0.1,␣
↪learning_rate = 0.1, epsilon_0 = 0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9014084507042254
fold 2 training in progress
fold 2 training completed, accuracy =  1.0
fold 3 training in progress
fold 3 training completed, accuracy =  0.9714285714285714
fold 4 training in progress
fold 4 training completed, accuracy =  0.9714285714285714
fold 5 training in progress
fold 5 training completed, accuracy =  0.9714285714285714
fold 6 training in progress
fold 6 training completed, accuracy =  0.9285714285714286
fold 7 training in progress
fold 7 training completed, accuracy =  0.9571428571428572
fold 8 training in progress
fold 8 training completed, accuracy =  1.0
fold 9 training in progress
fold 9 training completed, accuracy =  0.9855072463768116
fold 10 training in progress
fold 10 training completed, accuracy =  0.9420289855072463
```

[ ]:
```
epoch = 2000
hiddenlayerparemeter = [4]
accuracy3_5, precision3_5, recall3_5, fscore_cancer3_5=␣
↪meanevaluation(listoflistofoutputs3_5,1)
print("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc3_5)))
print("fscore:", float("{0:.4f}". format(fscore_cancer3_5)))
plt.plot(range(epoch+1), listofjlist3_5[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + "␣
↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Cancer Data Neural Network with [4] hidden layers and 2000 epochs
accuracy: 0.9629
fscore: 0.9472
```



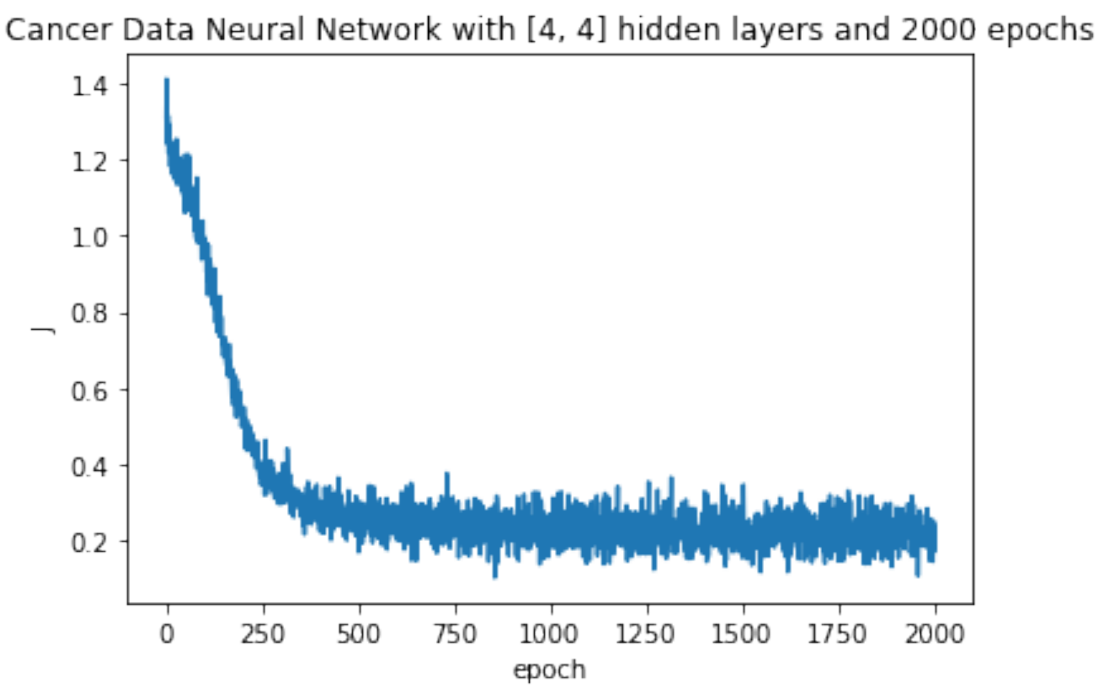
[ ]:
```
hiddenlayerparemeter = [4,4]
epoch = 2000
listoflistofoutputs3_6, acc3_6, listofjlist3_6 =␣
↪kfoldcrossvalidneuralnetwork(cancerdata, cancercategory,␣
↪hiddenlayerparemeter, k = 10, minibatchk = 3, lambda_reg = 0.1,␣
↪learning_rate = 0.1, epsilon_0 = 0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9436619718309859
fold 2 training in progress
fold 2 training completed, accuracy =  0.9714285714285714
fold 3 training in progress
fold 3 training completed, accuracy =  0.9857142857142858
fold 4 training in progress
fold 4 training completed, accuracy =  0.9714285714285714
fold 5 training in progress
fold 5 training completed, accuracy =  0.9714285714285714
fold 6 training in progress
fold 6 training completed, accuracy =  0.9714285714285714
fold 7 training in progress
fold 7 training completed, accuracy =  0.9285714285714286
fold 8 training in progress
fold 8 training completed, accuracy =  0.9571428571428572
fold 9 training in progress
```

```
fold 9 training completed, accuracy =  0.9710144927536232
fold 10 training in progress
fold 10 training completed, accuracy =  0.9420289855072463
```

[ ]:
```python
accuracy3_6, precision3_6, recall3_6, fscore_cancer3_6=␣
    ↪meanevaluation(listoflistofoutputs3_6,1)
print("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
    ↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc3_6)))
print("fscore:", float("{0:.4f}". format(fscore_cancer3_6)))
plt.plot(range(epoch+1), listofjlist3_6[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + "␣
    ↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Cancer Data Neural Network with [4, 4] hidden layers and 2000 epochs
accuracy: 0.9614
fscore: 0.9448
```



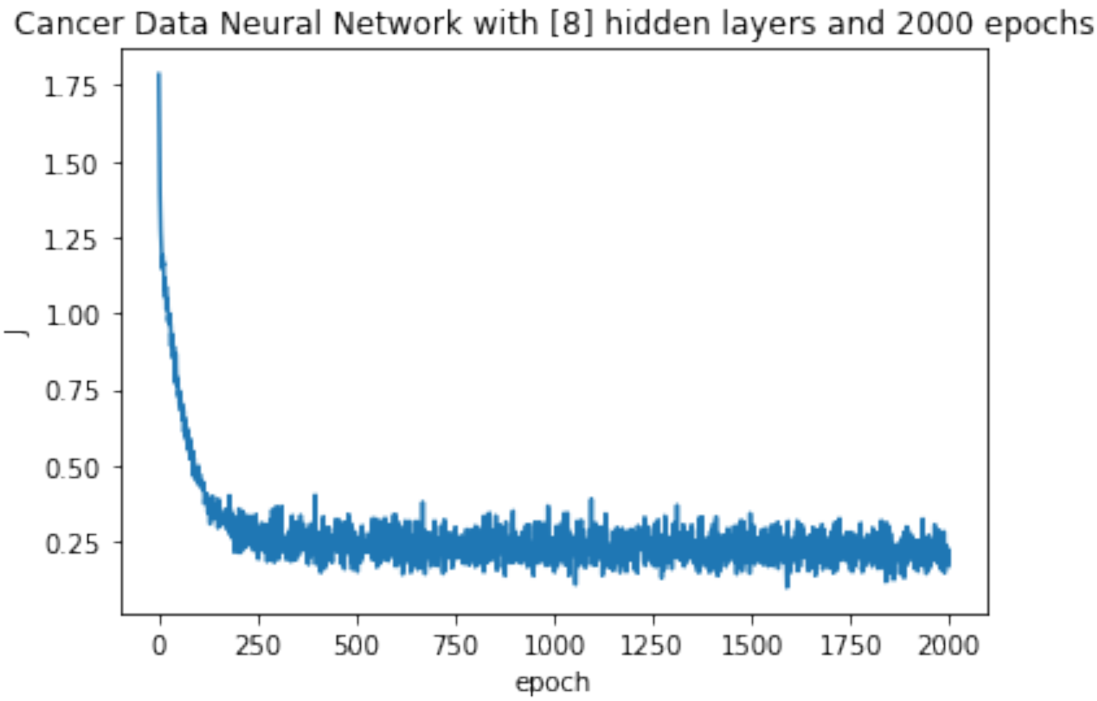
[ ]:
```python
hiddenlayerparemeter = [8]
epoch = 2000
```

```
listoflistofoutputs3_7, acc3_7, listofjlist3_7 =␣
 ↪kfoldcrossvalidneuralnetwork(cancerdata, cancercategory,␣
 ↪hiddenlayerparemeter, k = 10, minibatchk = 3, lambda_reg = 0.1,␣
 ↪learning_rate = 0.1, epsilon_0 = 0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.9859154929577465
fold 2 training in progress
fold 2 training completed, accuracy =  0.9571428571428572
fold 3 training in progress
fold 3 training completed, accuracy =  0.9
fold 4 training in progress
fold 4 training completed, accuracy =  0.9857142857142858
fold 5 training in progress
fold 5 training completed, accuracy =  0.9714285714285714
fold 6 training in progress
fold 6 training completed, accuracy =  0.9428571428571428
fold 7 training in progress
fold 7 training completed, accuracy =  0.9571428571428572
fold 8 training in progress
fold 8 training completed, accuracy =  0.9857142857142858
fold 9 training in progress
fold 9 training completed, accuracy =  0.9855072463768116
fold 10 training in progress
fold 10 training completed, accuracy =  0.9710144927536232
```

[ ]:
```
hiddenlayerparemeter = [8]
accuracy3_7, precision3_7, recall3_7, fscore_cancer3_7=␣
 ↪meanevaluation(listoflistofoutputs3_7,1)
print("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
 ↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc3_7)))
print("fscore:", float("{0:.4f}". format(fscore_cancer3_7)))
plt.plot(range(epoch+1), listofjlist3_7[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + "␣
 ↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Cancer Data Neural Network with [8] hidden layers and 2000 epochs
accuracy: 0.9642
fscore: 0.9489
```



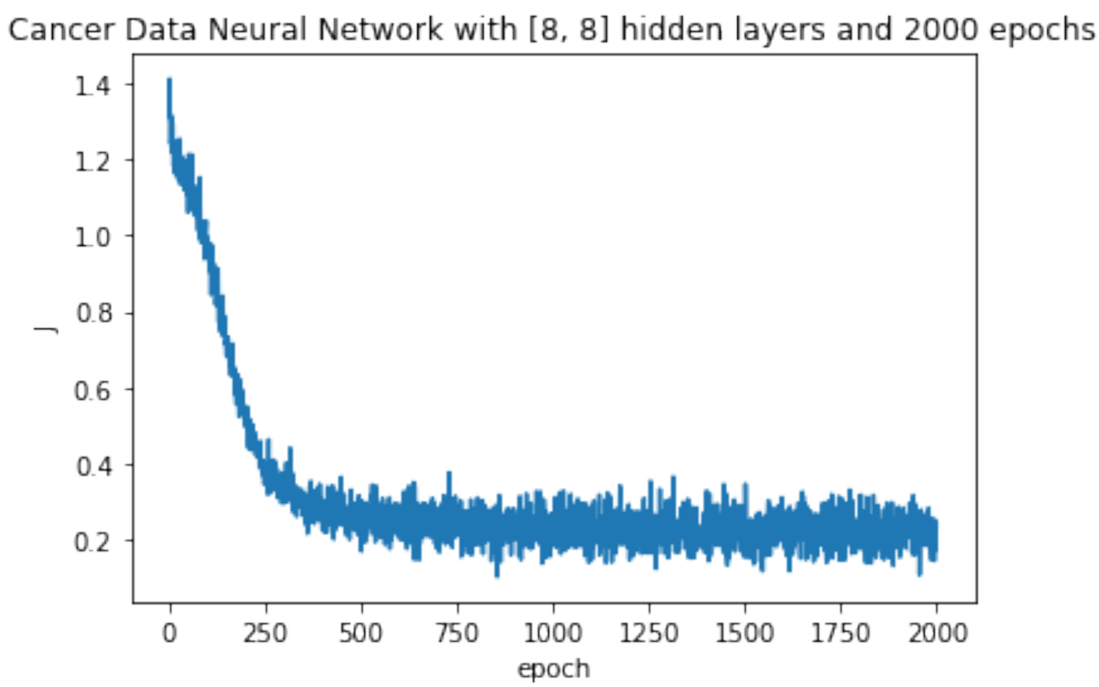

```
[ ]: hiddenlayerparemeter = [8,8]
epoch = 2000
listoflistofoutputs3_8, acc3_8, listofjlist3_8 =␣
 ↪kfoldcrossvalidneuralnetwork(cancerdata, cancercategory,␣
 ↪hiddenlayerparemeter, k = 10, minibatchk = 3, lambda_reg = 0.1,␣
 ↪learning_rate = 0.1, epsilon_0 = 0.0001, softstop = epoch, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.971830985915493
fold 2 training in progress
fold 2 training completed, accuracy =  0.9714285714285714
fold 3 training in progress
fold 3 training completed, accuracy =  0.9285714285714286
fold 4 training in progress
fold 4 training completed, accuracy =  0.9857142857142858
fold 5 training in progress
fold 5 training completed, accuracy =  0.9571428571428572
fold 6 training in progress
fold 6 training completed, accuracy =  0.9857142857142858
fold 7 training in progress
fold 7 training completed, accuracy =  0.9714285714285714
fold 8 training in progress
fold 8 training completed, accuracy =  0.9285714285714286
fold 9 training in progress
```

```
fold 9 training completed, accuracy =  0.9710144927536232
fold 10 training in progress
fold 10 training completed, accuracy =  0.9710144927536232
```

[ ]:
```python
accuracy3_8, precision3_8, recall3_8, fscore_cancer3_8=␣
 ↪meanevaluation(listoflistofoutputs3_6,1)
print("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + " hidden␣
 ↪layers and " + str(epoch) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc3_8)))
print("fscore:", float("{0:.4f}". format(fscore_cancer3_8)))
plt.plot(range(epoch+1), listofjlist3_6[1])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("Cancer Data Neural Network with " + str(hiddenlayerparemeter) + "␣
 ↪hidden layers and " + str(epoch) + " epochs")
plt.show()
```

```
Cancer Data Neural Network with [8, 8] hidden layers and 2000 epochs
accuracy: 0.9642
fscore: 0.9448
```



CANCER Data | **Hidden Layers** | **(2)LR=0.1** | **(2)LR=0.05** | **(2)LR=0.01** | **(2)LR=0.01** | **(4)LR=0.1** | **(4,4)LR=0.1** | **(8)LR=0.1** | **(8,8)LR=0.1** | | :— | :— | :— | :— | :— | :— | :— | :— | :— | | **Epoch** | 2000 | 2000 | 2000 | 4000 | 1000 | 2000 | 4000 | 4000 | | **Accuracy** | 0.9627 | 0.9614| 0.9571 | 0.9628 | 0.9629 | 0.9614 | 0.9642| 0.9642 | | **F-Score** | 0.9469| 0.9447 | 0.9365 | 0.9462 | 0.9472 | 0.9448 | 0.9489 | 0.9448 |

CANCER Data

Hidden Layers	(2)LR=0.1	(2)LR=0.05	(2)LR=0.01	(2)LR=0.01	(4)LR=0.1	(4,4)LR=0.1	(8)LR=0.1	(8,8)LR=0.1
Epoch	2000	2000	2000	4000	1000	2000	4000	4000
Accuracy	0.9627	0.9614	0.9571	0.9628	0.9629	0.9614	0.9642	0.9642
F-Score	0.9469	0.9447	0.9365	0.9462	0.9472	0.9448	0.9489	0.9448

Analysis of CANCER Data

> Discuss (on a high level) what contributed the most to improving performance: changing the regularization parameter; adding more layers; having deeper networks with many layers but few neurons per layer? designing networks with few layers but many neurons per layer? Discuss any patterns that you may have encountered. Also, discuss whether there is a point where constructing and training more "sophisticated"/complex networks—i.e., larger networks—no longer improves performance (or worsens performance).
>
> Based on the analyses above, discuss which neural network architecture you would select if you had to deploy such a classifier in real life. Explain your reasoning.

ANSWER: The overall performance is also pretty nice for this data set. What I've notice while plotting them is that I find out that once the j get smaller, it becomes very noisy, so here I am trying to reduce the noise by changing the learning rate for them.

What we can see here is that smaller learning rate did decrease the noise, but there are sill many left. Maybe heuristic is a solotion to it.

Also, it verified what I wrote in the Wine Data Analysis: The smaller learning rate requires more epochs to converge.

In real world, I think I will just use the [2], LR=1, 2000-epochs model, considering that cancer might have learning relation between the input to the output.

It's also a choice to analyze this using the ROC because in cancer, we want to find all real_positive/actual_positive.

1.2.3 EXTRA III: CMC Data

Trained with MiniBatch, vectorized neural network, divided to minibatchk group,

(I might modifiy the minibatchk(mbk), I will start with mbk = 15, that is batch size about 90 to 100.)

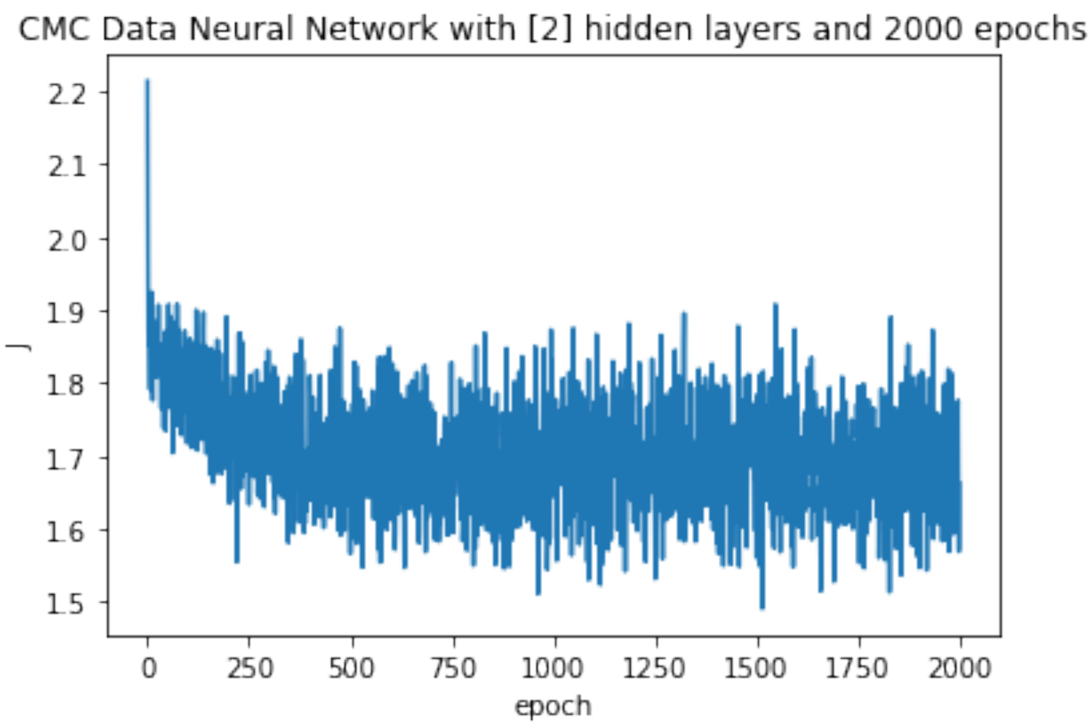
[ ]:
```
hiddenlayerparemeter4_1 = [2]
epoch4_1 = 2000
listoflistofoutputs4_1, acc4_1, listofjlist4_1 =␣
 ↪kfoldcrossvalidneuralnetwork(cmcdata, cmccategory, hiddenlayerparemeter4_1,␣
 ↪k = 10, minibatchk = 15, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 =␣
 ↪0.0001, softstop = epoch4_1, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.5771812080536913
fold 2 training in progress
fold 2 training completed, accuracy =  0.4391891891891892
fold 3 training in progress
fold 3 training completed, accuracy =  0.5472972972972973
fold 4 training in progress
fold 4 training completed, accuracy =  0.47619047619047616
fold 5 training in progress
```

```
fold 5 training completed, accuracy =  0.5306122448979592
fold 6 training in progress
fold 6 training completed, accuracy =  0.5170068027210885
fold 7 training in progress
fold 7 training completed, accuracy =  0.5170068027210885
fold 8 training in progress
fold 8 training completed, accuracy =  0.5102040816326531
fold 9 training in progress
fold 9 training completed, accuracy =  0.4965986394557823
fold 10 training in progress
fold 10 training completed, accuracy =  0.5068493150684932
```

[ ]:
```
accuracy4_1, precision4_1, recall4_1, fscore_cmc4_1=␣
  ↪meanevaluation(listoflistofoutputs4_1,1)
print("CMC Data Neural Network with " + str(hiddenlayerparemeter4_1) + " hidden␣
  ↪layers and " + str(epoch4_1) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc4_1)))
print("fscore:", float("{0:.4f}". format(fscore_cmc4_1)))
plt.plot(range(epoch4_1+1), listofjlist4_1[-3])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("CMC Data Neural Network with " + str(hiddenlayerparemeter4_1) + "␣
  ↪hidden layers and " + str(epoch4_1) + " epochs")
plt.show()
```

```
CMC Data Neural Network with [2] hidden layers and 2000 epochs
accuracy: 0.5118
fscore: 0.4092
```



The performance is about 0.5, which is the least want output. (even 0.1 would be better). Besides, this Doesn't converge to a good spot. I'd assume this is a local minimum.

The problem might be the learning rate: too low?

Also try different mbk for batchsize and more neuron.

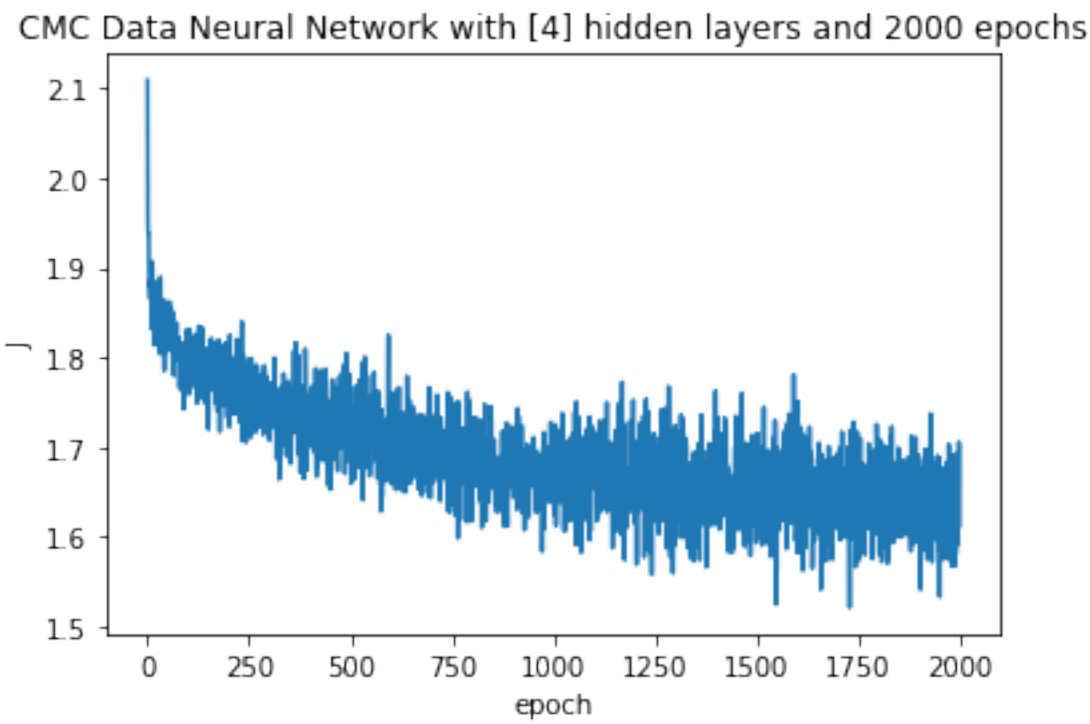
[ ]:
```
hiddenlayerparemeter4_2 = [4]
epoch4_2 = 2000
listoflistofoutputs4_2, acc4_2, listofjlist4_2 =␣
 ↪kfoldcrossvalidneuralnetwork(cmcdata, cmccategory, hiddenlayerparemeter4_2,␣
 ↪k = 10, minibatchk = 5, lambda_reg = 0.1, learning_rate = 0.13, epsilon_0 =␣
 ↪0.00001, softstop = epoch4_2, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.5838926174496645
fold 2 training in progress
fold 2 training completed, accuracy =  0.5202702702702703
fold 3 training in progress
fold 3 training completed, accuracy =  0.5067567567567568
fold 4 training in progress
fold 4 training completed, accuracy =  0.5578231292517006
fold 5 training in progress
fold 5 training completed, accuracy =  0.5102040816326531
fold 6 training in progress
```

```
fold 6 training completed, accuracy =  0.5306122448979592
fold 7 training in progress
fold 7 training completed, accuracy =  0.5034013605442177
fold 8 training in progress
fold 8 training completed, accuracy =  0.5170068027210885
fold 9 training in progress
fold 9 training completed, accuracy =  0.5034013605442177
fold 10 training in progress
fold 10 training completed, accuracy =  0.541095890410959
```

[ ]:
```
accuracy4_2, precision4_2, recall4_2, fscore_cmc4_2=␣
 ↪meanevaluation(listoflistofoutputs4_2,1)
print("CMC Data Neural Network with " + str(hiddenlayerparemeter4_2) + " hidden␣
 ↪layers and " + str(epoch4_2) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc4_2)))
print("fscore:", float("{0:.4f}". format(fscore_cmc4_2)))
plt.plot(range(epoch4_2+1), listofjlist4_2[0])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("CMC Data Neural Network with " + str(hiddenlayerparemeter4_2) + "␣
 ↪hidden layers and " + str(epoch4_2) + " epochs")
plt.show()
```

```
CMC Data Neural Network with [4] hidden layers and 2000 epochs
accuracy: 0.5274
fscore: 0.4278
```



Sadly, there's only small improvements. Both network kind of stuck around j = 1.7, maybe I will run the next network with more epochs, and try out more layers.. I will use kfold k = 5 instead of 10 so it won't take forever

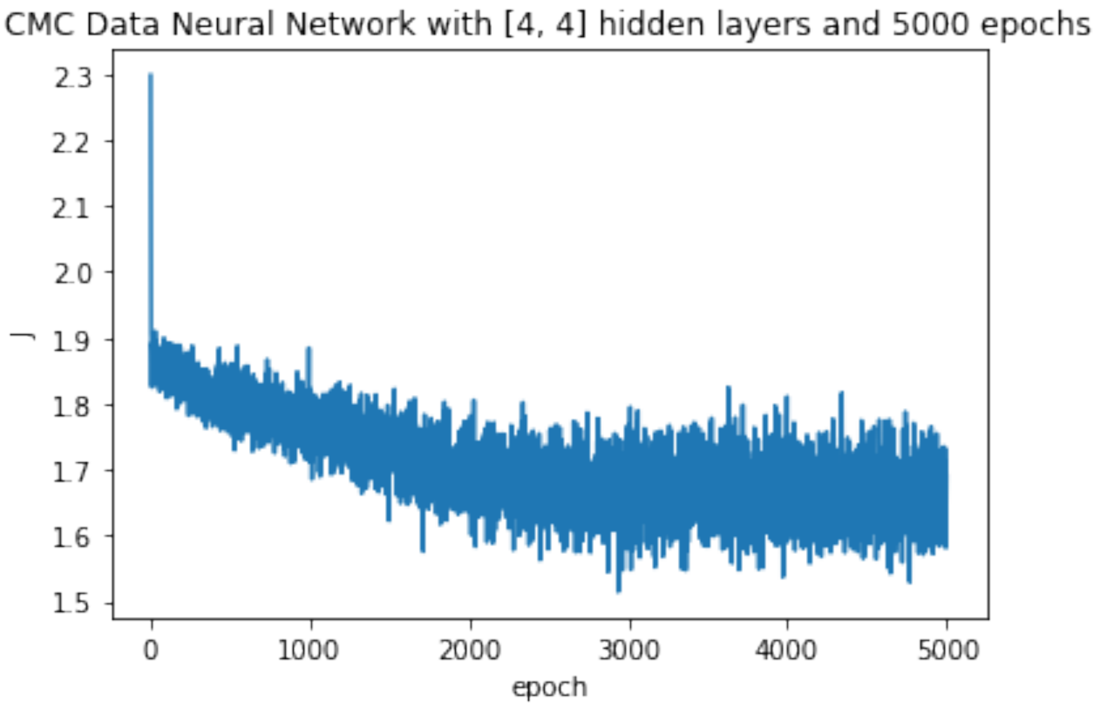
But maybe, from the graph, I'd guess it's just the minimum around 1.6?

```
[ ]: hiddenlayerparemeter4_3 = [4,4]
epoch4_3 = 5000
listoflistofoutputs4_3, acc4_3, listofjlist4_3 =␣
 ↪kfoldcrossvalidneuralnetwork(cmcdata, cmccategory, hiddenlayerparemeter4_3,␣
 ↪k = 5, minibatchk = 5, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
 ↪00001, softstop = epoch4_3, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.543918918918919
fold 2 training in progress
fold 2 training completed, accuracy =  0.576271186440678
fold 3 training in progress
fold 3 training completed, accuracy =  0.5389830508474577
fold 4 training in progress
fold 4 training completed, accuracy =  0.5238095238095238
fold 5 training in progress
fold 5 training completed, accuracy =  0.4812286689419795
```

```
[ ]: accuracy4_3, precision4_3, recall4_3, fscore_cmc4_3=␣
 ↪meanevaluation(listoflistofoutputs4_3,1)
print("CMC Data Neural Network with " + str(hiddenlayerparemeter4_3) + " hidden␣
 ↪layers and " + str(epoch4_3) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc4_3)))
print("fscore:", float("{0:.4f}". format(fscore_cmc4_3)))
plt.plot(range(epoch4_3+1), listofjlist4_3[0])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("CMC Data Neural Network with " + str(hiddenlayerparemeter4_3) + "␣
 ↪hidden layers and " + str(epoch4_3) + " epochs")
plt.show()
```

```
CMC Data Neural Network with [4, 4] hidden layers and 5000 epochs
accuracy: 0.5328
fscore: 0.4197
```



Still Not a lot of changes.

(1) Try Higher learning rate (0.2)

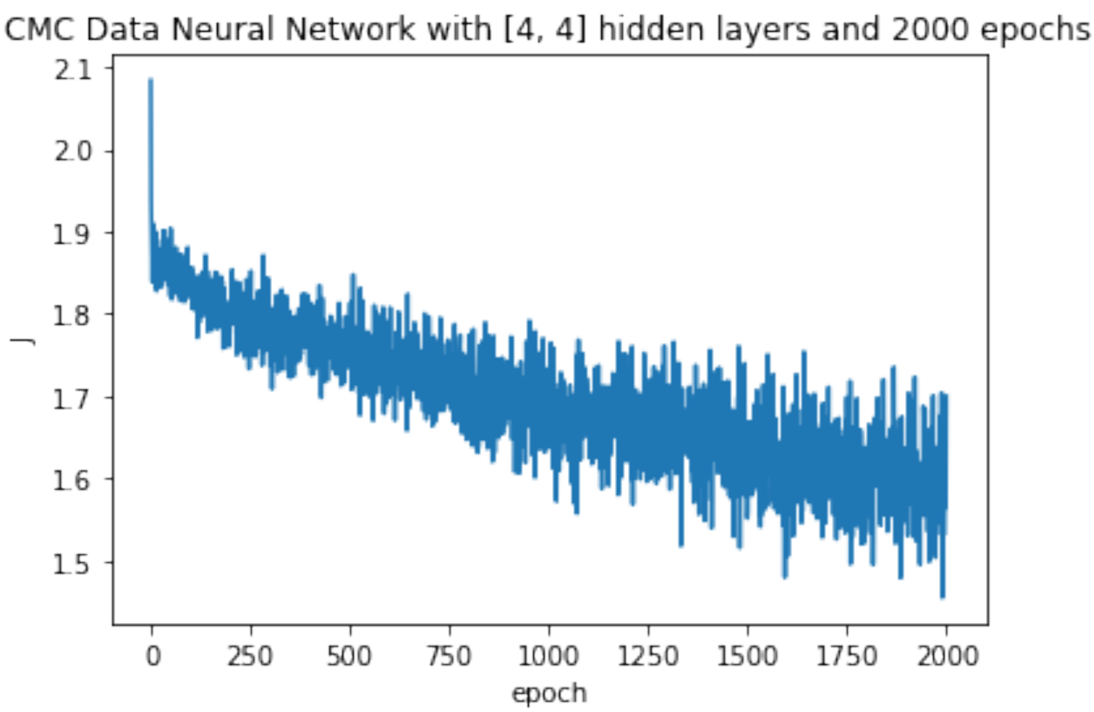
(2) Try More layers [4,4,4],[8,8,8,8]

[ ]:
```
hiddenlayerparemeter4_4 = [4,4]
epoch4_4 = 2000
listoflistofoutputs4_4, acc4_4, listofjlist4_4 =␣
↪kfoldcrossvalidneuralnetwork(cmcdata, cmccategory, hiddenlayerparemeter4_4,␣
↪k = 5, minibatchk = 5, lambda_reg = 0.1, learning_rate = 0.2, epsilon_0 = 0.
↪00001, softstop = epoch4_4, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.5777027027027027
fold 2 training in progress
fold 2 training completed, accuracy =  0.5796610169491525
fold 3 training in progress
fold 3 training completed, accuracy =  0.5898305084745763
fold 4 training in progress
fold 4 training completed, accuracy =  0.4897959183673469
fold 5 training in progress
fold 5 training completed, accuracy =  0.5290102389078498
```

[ ]:
```
accuracy4_4, precision4_4, recall4_4, fscore_cmc4_4=␣
 ↪meanevaluation(listoflistofoutputs4_4,1)
print("CMC Data Neural Network with " + str(hiddenlayerparemeter4_4) + " hidden␣
 ↪layers and " + str(epoch4_4) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc4_4)))
print("fscore:", float("{0:.4f}". format(fscore_cmc4_4)))
plt.plot(range(epoch4_4+1), listofjlist4_4[0])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("CMC Data Neural Network with " + str(hiddenlayerparemeter4_4) + "␣
 ↪hidden layers and " + str(epoch4_4) + " epochs")
plt.show()
```

```
CMC Data Neural Network with [4, 4] hidden layers and 2000 epochs
accuracy: 0.5532
fscore: 0.4284
```



The sign of convergance! Try larger epoch

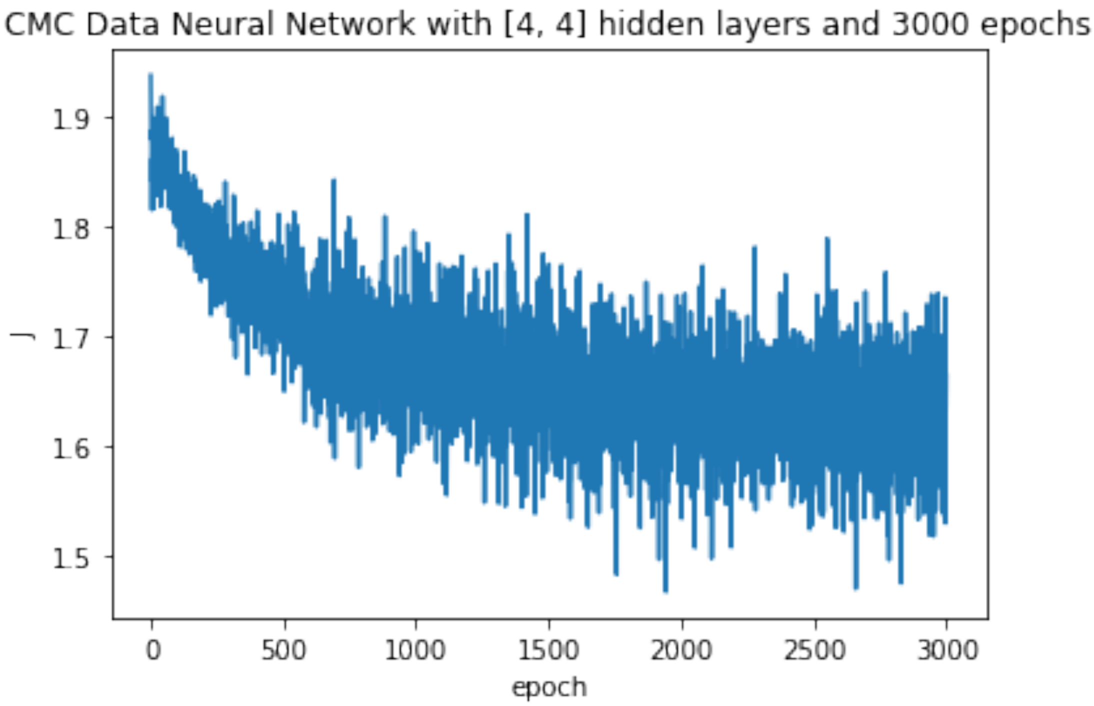
[ ]:
```
hiddenlayerparemeter4_5 = [4,4]
epoch4_5 = 3000
```

```
listoflistofoutputs4_5, acc4_5, listofjlist4_5 =␣
↪kfoldcrossvalidneuralnetwork(cmcdata, cmccategory, hiddenlayerparemeter4_5,␣
↪k = 5, minibatchk = 5, lambda_reg = 0.1, learning_rate = 0.22, epsilon_0 = 0.
↪00001, softstop = epoch4_5, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.5608108108108109
fold 2 training in progress
fold 2 training completed, accuracy =  0.5423728813559322
fold 3 training in progress
fold 3 training completed, accuracy =  0.5559322033898305
fold 4 training in progress
fold 4 training completed, accuracy =  0.5578231292517006
fold 5 training in progress
fold 5 training completed, accuracy =  0.5392491467576792
```

[ ]:
```
accuracy4_5, precision4_5, recall4_5, fscore_cmc4_5=␣
↪meanevaluation(listoflistofoutputs4_5,1)
print("CMC Data Neural Network with " + str(hiddenlayerparemeter4_5) + " hidden␣
↪layers and " + str(epoch4_5) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc4_5)))
print("fscore:", float("{0:.4f}". format(fscore_cmc4_5)))
plt.plot(range(epoch4_5+1), listofjlist4_5[0])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("CMC Data Neural Network with " + str(hiddenlayerparemeter4_5) + "␣
↪hidden layers and " + str(epoch4_5) + " epochs")
plt.show()
```

```
CMC Data Neural Network with [4, 4] hidden layers and 3000 epochs
accuracy: 0.5512
fscore: 0.3968
```



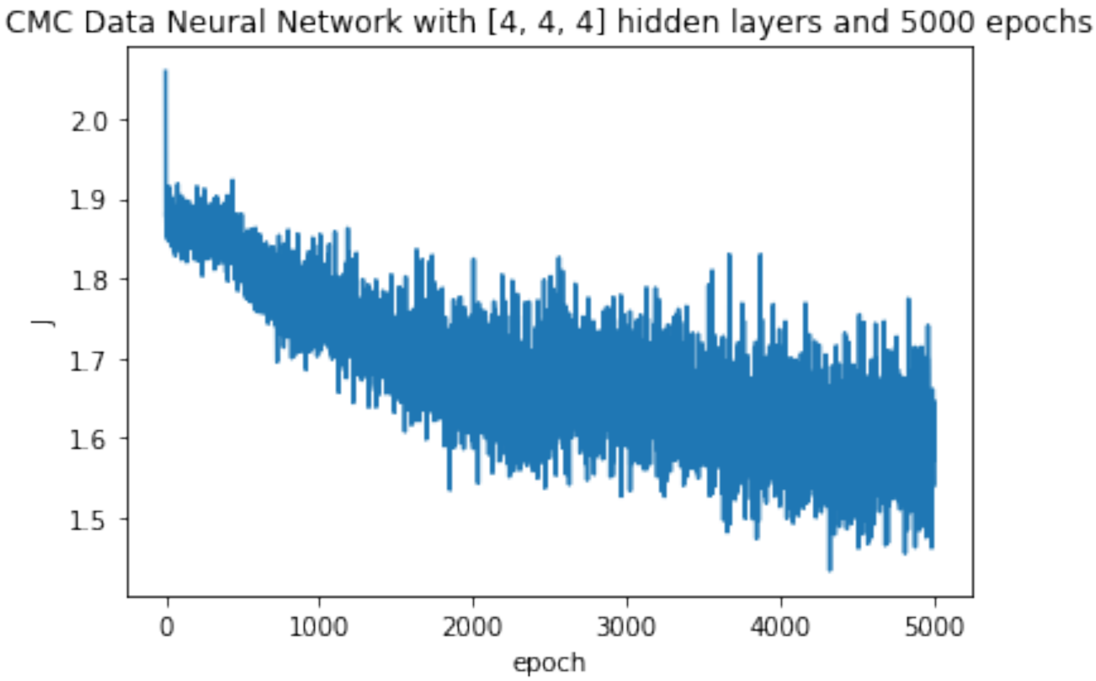

```
[ ]: hiddenlayerparemeter4_6 = [4,4,4]
     epoch4_6 = 5000
     listoflistofoutputs4_6, acc4_6, listofjlist4_6 =␣
      ↪kfoldcrossvalidneuralnetwork(cmcdata, cmccategory, hiddenlayerparemeter4_6,␣
      ↪k = 5, minibatchk = 5, lambda_reg = 0.1, learning_rate = 0.2, epsilon_0 = 0.
      ↪00001, softstop = epoch4_6, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.5371621621621622
fold 2 training in progress
fold 2 training completed, accuracy =  0.5423728813559322
fold 3 training in progress
fold 3 training completed, accuracy =  0.5423728813559322
fold 4 training in progress
fold 4 training completed, accuracy =  0.5374149659863946
fold 5 training in progress
fold 5 training completed, accuracy =  0.5563139931740614
```

```
[ ]: accuracy4_6, precision4_6, recall4_6, fscore_cmc4_6=␣
      ↪meanevaluation(listoflistofoutputs4_6,1)
     print("CMC Data Neural Network with " + str(hiddenlayerparemeter4_6) + " hidden␣
      ↪layers and " + str(epoch4_6) + " epochs")
     print("accuracy:", float("{0:.4f}". format(acc4_6)))
     print("fscore:", float("{0:.4f}". format(fscore_cmc4_6)))
```

```
plt.plot(range(epoch4_6+1), listofjlist4_6[0])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("CMC Data Neural Network with " + str(hiddenlayerparemeter4_6) + "␣
↪hidden layers and " + str(epoch4_6) + " epochs")
plt.show()
```

```
CMC Data Neural Network with [4, 4, 4] hidden layers and 5000 epochs
accuracy: 0.5431
fscore: 0.3392
```



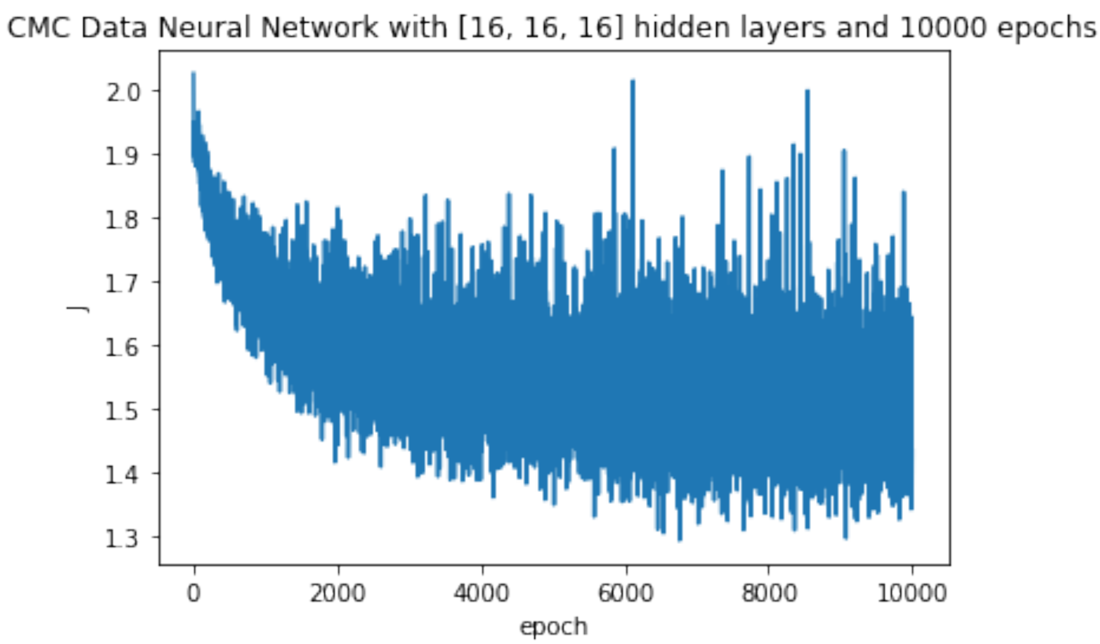
[ ]:
```
hiddenlayerparemeter4_7 = [16,16,16]
epoch4_7 = 10000
listoflistofoutputs4_7, acc4_7, listofjlist4_7 =␣
↪kfoldcrossvalidneuralnetwork(cmcdata, cmccategory, hiddenlayerparemeter4_7,␣
↪k = 5, minibatchk = 5, lambda_reg = 0.1, learning_rate = 0.2, epsilon_0 = 0.
↪00001, softstop = epoch4_7, printq = False)
```

```
fold 1 training in progress
fold 1 training completed, accuracy =  0.4797297297297297
fold 2 training in progress
fold 2 training completed, accuracy =  0.511864406779661
fold 3 training in progress
fold 3 training completed, accuracy =  0.5050847457627119
fold 4 training in progress
```

```
fold 4 training completed, accuracy =  0.5034013605442177
fold 5 training in progress
fold 5 training completed, accuracy =  0.5085324232081911
```

[ ]:
```
accuracy4_7, precision4_7, recall4_7, fscore_cmc4_7=␣
 ↪meanevaluation(listoflistofoutputs4_7,1)
print("CMC Data Neural Network with " + str(hiddenlayerparemeter4_7) + " hidden␣
 ↪layers and " + str(epoch4_7) + " epochs")
print("accuracy:", float("{0:.4f}". format(acc4_7)))
print("fscore:", float("{0:.4f}". format(fscore_cmc4_7)))
plt.plot(range(epoch4_7+1), listofjlist4_7[0])
plt.xlabel("epoch")
plt.ylabel("J")
plt.title("CMC Data Neural Network with " + str(hiddenlayerparemeter4_7) + "␣
 ↪hidden layers and " + str(epoch4_7) + " epochs")
plt.show()
```

```
CMC Data Neural Network with [16, 16, 16] hidden layers and 10000 epochs
accuracy: 0.5017
fscore: 0.4065
```



So that seems to demonstrate we can have min j around 1.4, but I don't exactly know the tuning, maybe I will give up on find a better tuning for this data.

CMC Data | **Hidden Layers** | **(2)LR=0.1** | **(4)LR=0.1** | **(4,4)LR=0.1** | **(4,4)LR=0.2** | **(4,4)LR=0.22** | **(4,4,4)LR=0.2** | **(16,16,16)LR=0.2** | | :— | :— | :— | :— | :— | :— | :— | :— | | **Epoch** | 2000 | 2000 | 5000 | 2000 | 3000 | 5000 | 10000 | | **Accuracy** | 0.5118 | 0.5274 | 0.5328

CMC Data

Hidden Layers	(2)LR=0.1	(4)LR=0.1	(4,4)LR=0.1	(4,4)LR=0.2	(4,4)LR=0.22	(4,4,4)LR=0.2	(16,16,16)LR=0.2
Epoch	2000	2000	5000	2000	3000	5000	10000
Accuracy	0.5118	0.5274	0.5328	0.5532	0.5512	0.5431	0.5017
F-Score	0.4092	0.4278	0.4197	0.4284	0.3968	0.3392	0.4065

| 0.5532 | 0.5512 | 0.5431 | 0.5017 | | **F-Score** | 0.4092 | 0.4278 | 0.4197 | 0.4284 | 0.3968 | 0.3392 | 0.4065 |

1.2.4 EXTRA IV: Numerical Check Gradient

I will not do this.

1.3 APPENDIX

1.3.1 utils.py

[ ]:
```
from sqlite3 import Row
from evaluationmatrix import *
from sklearn import datasets
import sklearn.model_selection
from sklearn.preprocessing import OneHotEncoder
import random
import numpy as np
import csv
import math
import matplotlib.pyplot as plt
from collections import Counter

def importfile(name:str,delimit:str):
    # importfile('hw3_wine.csv', '\t')
    file = open("datasets/"+name, encoding='utf-8-sig')
    reader = csv.reader(file, delimiter=delimit)
    dataset = []
    for row in reader:
        dataset.append(row)
    file.close()
    return dataset

def onehotencoder(data,category):
    dataT=data.T.copy()
    enc = OneHotEncoder(sparse=False)
    i = 0
    appendeddict = {}
    for cat in category:
        if category[cat] == 'categorical':
            hotneeded = dataT[i]
            hotted = enc.fit_transform(hotneeded.reshape(-1,1))
            for j in enc.categories_[0]:
                newname = cat+'_'+str(j)
                appendeddict[newname] = 'ohe_numerical'
            dataT = np.append(dataT,hotted.T,axis=0)
        if category[cat] == 'class':
            hotneeded = dataT[i]
```

```
            hotted = enc.fit_transform(hotneeded.reshape(-1,1))
            for class_ in enc.categories_[0]:
                newname = cat+'_'+str(class_)
                appendeddict[newname] = 'class_numerical'
            dataT = np.append(dataT,hotted.T,axis=0)
        i += 1

    category.update(appendeddict)
    i = 0
    categorycopy = category.copy()
    droplist = []
    for cat in category:
        if category[cat] == 'categorical' or category[cat] == 'class':
            droplist.append(i)
            categorycopy.pop(cat)
        i += 1
    dataT = np.delete(dataT,droplist,axis=0)
    return dataT.T, categorycopy

def normalizetrain(ohe_traindata, category): # input data in by row/by instance
    dataTC = ohe_traindata.T.copy()
    minmaxes = []
    i = 0
    for oneattribute in category:
        if category[oneattribute] == 'numerical':
            colmin = np.min(dataTC[i])
            colmax = np.max(dataTC[i])
            singleminmax = [colmin,colmax]
            # normalize to 0 to 1
            for j in range(len(dataTC[i])):
                dataTC[i][j] = (dataTC[i][j] - colmin)/(colmax - colmin)
            minmaxes.append(singleminmax)
        else:
            minmaxes.append([0.0,1.0])
        i=i+1
    return dataTC.T, minmaxes

def normalizeonetest(instance_data, category, minmaxes):
    i = 0
    for oneattribute in category:
        if category[oneattribute] == 'numerical':
            instance_data[i] = (instance_data[i] - minmaxes[i][0])/
 ↪(minmaxes[i][1] - minmaxes[i][0])
        i += 1
    return instance_data

def normalizealltest(ohe_testdata, category, minmaxes):
```

```
    result = []
    for i in ohe_testdata:
        n_ohe_test = normalizeonetest(i, category, minmaxes)
        result.append(n_ohe_test)
    return np.array(result)

def g(x): # sigmoid function
    return 1/(1 + np.exp(-x))

def transposelistoflist(l):
    newlistoflist = []
    for i in range(len(l[0])):
        newlist = []
        for j in range(len(l)):
            newlist.append(l[j][i])
        newlistoflist.append(newlist)
    return newlistoflist
```

1.3.2 neuralnetwork.py

[ ]:
```
from utils import *

def initialize_weights(ohe_category,layer_parameter, biasterm=True):
    weight_matrix_list = []

    inputcategory, outputcategory = [],[]
    inputindex, outputindex = [],[]
    n = 0
    for i in ohe_category:
        if ohe_category[i] != 'class_numerical':
            inputcategory.append(i) # name of the input category
            inputindex.append(n) # index of the input category
        else:
            outputcategory.append(i) # name of the output category
            outputindex.append(n) # index of the output category
        n += 1

    b = 1 if biasterm == True else 0

    updatedlayerparameterwbias = [len(inputcategory)+b] + list(np.
 ↪array(layer_parameter)+b) + [len(outputcategory)] # [inputlayer,␣
 ↪layerparameters, outputlayer]
    for i in range(len(updatedlayerparameterwbias)-1):
        layernow = updatedlayerparameterwbias[i]
        layernext = updatedlayerparameterwbias[i+1]-1 if i !
 ↪=len(updatedlayerparameterwbias)-2 else updatedlayerparameterwbias[i+1]
```

```
        # ^ for the last layer, the bias is not included, so don't need to␣
↪minus 1 ^
        init_weight = np.random.rand(layernext,layernow) * 2 - 1 # initialize␣
↪the weight with random number between -1 and 1
        weight_matrix_list.append(init_weight)

    return weight_matrix_list

def costfunction(expected_output, actual_output):
    j = -np.multiply(expected_output,np.log(actual_output)) - np.multiply((1 -␣
↪expected_output),np.log(1 - actual_output))
    return np.sum(j)

def sumofweights(listofweights,bias=True): # computes the square of all weights␣
↪of the network and sum them up
    sum = 0
    for weight in listofweights:
        if bias:
            w = weight.copy()
            w[:, 0] = 0
            sum += np.sum(np.square(w))
        else:
            sum += np.sum(np.square(weight))
    return sum

def blame(predict_output, expected_output, weights_list, a_list, biasterm=True):
↪ # This is to find out the delta function
    deltalist = []
    delta_layer_l = predict_output - expected_output
    deltalist.append(delta_layer_l)
    i = len(weights_list)-1
    current_delta = delta_layer_l

    while i > 0:
        delta_layer_now = np.multiply(np.multiply(np.dot(weights_list[i].
↪T,current_delta),a_list[i]),(1-a_list[i]))
        if biasterm:
            delta_layer_now[0] = 1 # the first attribute is the bias
            current_delta = delta_layer_now[1:] # the first attribute is the␣
↪bias
        else:
            current_delta = delta_layer_now
        deltalist.append(current_delta)
        i-=1
    deltalist.reverse()
```

```
    return deltalist

def gradientD(weights_list,deltalist,attributelist,biasterm=True):
    gradlist = []
    for i in range(len(weights_list)):
        attributenow = attributelist[i]
        deltanow = np.array([deltalist[i]]).T
        dotproduct = deltanow*attributenow
        # print('dotshape',dotproduct.shape)
        gradlist.append(dotproduct)
    return gradlist

# Forward propagation vectorized
def neural_network(normed_hotted_data,ohe_category,weights_list, minibatchk =␣
 ↪15, lambda_reg = 0.2, learning_rate = 0.01):
    biasterm=True
    normed_ohe_copy = normed_hotted_data.copy()
    if minibatchk > len(normed_hotted_data):
        minibatchk = len(normed_hotted_data)
    np.random.shuffle(normed_ohe_copy)
    # print('minibatchk',minibatchk)
    # print('shape of normed_ohe_copy',normed_ohe_copy.shape)
    splitted = np.array_split(normed_ohe_copy, minibatchk)

    inputcategory, outputcategory = [],[]
    inputindex, outputindex = [],[]
    n = 0
    for i in ohe_category:
        if ohe_category[i] != 'class_numerical':
            inputcategory.append(i) # name of the input category
            inputindex.append(n) # index of the input category
        else:
            outputcategory.append(i) # name of the output category
            outputindex.append(n) # index of the output category
        n += 1

    b = 1 if biasterm else 0

    for onebatch in splitted:
        onebatch = onebatch.T
        input_data = onebatch[inputindex].T
        output_data = onebatch[outputindex].T
        # input_data_mean = onebatch[inputindex].mean(axis=1)
        output_data = onebatch[outputindex].T

        # forward propagation
        instance_index = 0
```

```
        j = 0
        listofgradient = []
        for one_instance in input_data:
            current_layer_a = np.append(1,one_instance) if b == 1 else␣
↪one_instance
            # input layer is the current layer
            current_layer_index = 0
            output_expect = output_data[instance_index]
            attributesnobias = [one_instance]
            attributeswbias = [current_layer_a]
            for theta in weights_list:
                z = np.dot(theta,current_layer_a)
                a = g(z)
                current_layer_a = np.append(1,a) if (b == 1) and␣
↪(current_layer_index+1 != len(weights_list)) else a
                attributesnobias.append(a)
                attributeswbias.append(current_layer_a)
                current_layer_index += 1

            output_predict = current_layer_a # the last attribute is the output␣
↪for this batch.
            instance_index += 1
            j += costfunction(output_expect,output_predict)

            # calculate delta blame (back propagation)
            listofdelta = blame(output_predict,output_expect,weights_list,␣
↪attributeswbias)
            thisgradient =␣
↪gradientD(weights_list,listofdelta,attributeswbias,biasterm)
            listofgradient.append(thisgradient)

        gradientP = [lambda_reg*t for t in weights_list]
        # first column in singleP in the np.array = 0
        for singleP in gradientP:
            singleP[:, 0] = 0

        grad_D_transpose = transposelistoflist(listofgradient)
        grad_D_sum = [np.sum(t,axis=0) for t in grad_D_transpose]
        gradients_batch = []
        for i in range(len(grad_D_sum)):
            gradients_batch.append((grad_D_sum[i] + gradientP[i])*(1/
↪instance_index))

        j /= (instance_index+1)
        s = sumofweights(weights_list,bias=b)*lambda_reg/(2*(instance_index+1))
        allj = j+s # total cose with regularization
```

```
        # update weights
        for i in range(len(weights_list)):
            weights_list[i] -= learning_rate*gradients_batch[i]

    return weights_list, allj, j #j is j without regularization: only for␣
 ↪testing

def train_neural_network(normed_ohetraining_data,ohe_category,layerparameter,␣
 ↪minibatchk = 15, lambda_reg = 0.15, learning_rate = 0.01, epsilon_0 = 0.
 ↪00001, softstop = 8000, printq = False):
    init_weight = initialize_weights(ohe_category,layerparameter)
    updated_weight, jsum, purej =␣
 ↪neural_network(normed_ohetraining_data,ohe_category,init_weight, minibatchk,␣
 ↪lambda_reg, learning_rate)
    epsilon = epsilon_0 + 20
    currentj = jsum
    smallestj = jsum
    count = 0
    jlist = []
    jlist.append(currentj)
    while ((epsilon > epsilon_0) or (count < softstop) or (currentj >=␣
 ↪smallestj)) and (count < (softstop)):
        if printq:
            print('currentj',currentj)
            print('count',count)
        count += 1
        updated_weight, jsum, purej =␣
 ↪neural_network(normed_ohetraining_data,ohe_category,updated_weight,minibatchk,lambda_reg,le
        epsilon = jsum - currentj
        currentj = jsum
        jlist.append(currentj)
        if currentj < smallestj:
            smallestj = currentj

    return updated_weight, jlist

def predictoneinstance(inputdata,weightl): # inputdata here doesn't include the␣
 ↪class and bias.
    current_layer_a = np.append(1,inputdata)
    current_layer_index = 0
    alist = []
    alist.append(current_layer_a)
    for theta in weightl:
        z = np.dot(theta,current_layer_a)
        a = g(z)
```

```
        current_layer_a = np.append(1,a) if (current_layer_index+1 !=␣
↪len(weightl)) else a
        alist.append(current_layer_a)
        current_layer_index += 1
        raw_output = a
    predict_output = current_layer_a

    if len(predict_output) <=1:
        predict_output[0] = 0 if predict_output[0] <= 0.5 else 1
    else:
        predict_output[np.where(predict_output==np.max(predict_output))] = 1
        predict_output[np.where(predict_output!=1)] = 0

    return predict_output, raw_output

def predict_many_nn(testdatafull, ohecategory, weight):
    n = 0
    inputindex, outputindex = [],[]
    for i in ohecategory:
        if ohecategory[i] != 'class_numerical':
            inputindex.append(n)
        else:
            outputindex.append(n)
        n += 1
    predictvsexpectlist = [] # list of list of predict and expect/actual

    for instance in testdatafull:
        datainput = instance[inputindex]
        expect_output = instance[outputindex]
        predict_output, raw_output = predictoneinstance(datainput,weight)
        # process the index of value 1 in np.array
        processdexpect = np.where(expect_output==1)[0][0]
        processdpredict = np.where(predict_output==1)[0][0]
        predictvsexpectlist.append([processdpredict,processdexpect])

    correct = 0
    for outputtup in predictvsexpectlist:
        if outputtup[0] == outputtup[1]:
            correct += 1
    accuracy = correct/len(predictvsexpectlist)

    return predictvsexpectlist, accuracy
```

1.3.3 example.py

[ ]:
```python
import numpy as np
from utils import *
from stratified import *
from neuralnetwork import *

def g(x): # sigmoid function
    return 1/(1 + np.exp(-x))

# Theta 1
# 0.40000  0.10000
# 0.30000  0.20000
theta1 = np.array([[0.4, 0.1],[0.3,0.2]])
# Theta 2
#   0.70000  0.50000  0.60000
theta2 = np.array([0.7,0.5,0.6])
weightlist1= [theta1,theta2]

# Training set
#         Training instance 1
#                 x: [0.13000]
#                 y: [0.90000]
#         Training instance 2
#                 x: [0.42000]
#                 y: [0.23000]
# Training instance 1
trainingcategory = {'x1':'numerical', 'y':'class_numerical'}
trainingdata1 = np.array([0.13,0.9])
trainingdata2 = np.array([0.42,0.23])
inputdata1 = np.append(1,trainingdata1[0])
inputdata2 = np.append(1,trainingdata2[0])
exceptout1 = trainingdata1[1]
exceptout2 = trainingdata2[1]
lambda1 = 0

def costfunction(expected_output, actual_output):
    j = -np.multiply(expected_output,np.log(actual_output)) - np.multiply((1 -
 ↪expected_output),np.log(1 - actual_output))
    return np.sum(j)

def forwardtest(inputdata,weightl,expectedout):
    current_layer_a = inputdata
    print('current_a at 1 is',current_layer_a)
    current_layer_index = 0
    alist = []
    alist.append(current_layer_a)
```

```
    for theta in weightl:
        z = np.dot(theta,current_layer_a)
        a = g(z)
        current_layer_a = np.append(1,a) if (current_layer_index+1 !=␣
↪len(weightl)) else a
        print('current_a at',current_layer_index+2,'is',current_layer_a)
        alist.append(current_layer_a)
        current_layer_index += 1
    result = current_layer_a
    print('prediction is', result)
    print('exceptout is', expectedout)
    print('cost is', costfunction(expectedout,result))
    return result, costfunction(expectedout,result), alist


print('example 1 instance 1 forward test')
r1,j1,a1 = forwardtest(inputdata1,weightlist1,exceptout1)
# Computing the error/cost, J, of the network
#         Processing training instance 1
#         Forward propagating the input [0.13000]
#                 a1: [1.00000   0.13000]

#                 z2: [0.41300   0.32600]
#                 a2: [1.00000   0.60181   0.58079]

#                 z3: [1.34937]
#                 a3: [0.79403]

#                 f(x): [0.79403]
#         Predicted output for instance 1: [0.79403]
#         Expected output for instance 1: [0.90000]
#         Cost, J, associated with instance 1: 0.366
print('\n')

print('example 1 instance 2 forward test')
r2,j2,a2 = forwardtest(inputdata2,weightlist1,exceptout2)
        # Processing training instance 2
        # Forward propagating the input [0.42000]
        #         a1: [1.00000   0.42000]

        #         z2: [0.44200   0.38400]
        #         a2: [1.00000   0.60874   0.59484]

        #         z3: [1.36127]
        #         a3: [0.79597]

        #         f(x): [0.79597]
```

```
        # Predicted output for instance 2: [0.79597]
        # Expected output for instance 2: [0.23000]
        # Cost, J, associated with instance 2: 1.276
print('\n')

jlist1 = np.array([j1,j2])
numberofinstance1 = 2

def overallcost(jlist,n,weightl,lambda_reg):
    s = sumofweights(weightl,bias=0)*lambda_reg/(2*n)
    jsum = np.sum(jlist)
    return jsum/n + s

print('example 1 overall cost')
print(overallcost(jlist1,numberofinstance1,weightlist1,lambda1))
# Final (regularized) cost, J, based on the complete training set: 0.82098
print('\n')

def delta(weightl,alist,expect,actual):
    delta_layer_n = actual-expect
    deltalist = []
    deltalist.append(delta_layer_n)
    i = len(weightl)-1
    current_delta = delta_layer_n
    while i > 0:
        delta_layer_now = np.multiply(np.multiply(np.dot(weightl[i].
 ↪T,current_delta),alist[i]),(1-alist[i]))
        current_delta = delta_layer_now[1:]
        deltalist.append(current_delta)
        i-=1
    deltalist.reverse()
    return deltalist

def gradientD(weights_list,delta_list,a_list,biasterm=True):
    gradlist = []
    for i in range(len(weights_list)):
        anow = a_list[i]
        deltanow = np.array([delta_list[i]]).T
        dotproduct = deltanow*anow
        # print('dotshape',dotproduct.shape)
        gradlist.append(dotproduct)
    return gradlist

print('example 1 instance 1 theta value')
delta1_1 = delta(weightlist1,a1,exceptout1,r1)
        # Computing gradients based on training instance 1
        #         delta3: [-0.10597]
```

```
        #         delta2: [-0.01270   -0.01548]
print(delta1_1)
print('\n')

print('example 1 instance 1 gradient value')
                # Gradients of Theta2 based on training instance 1:
                #         -0.10597  -0.06378  -0.06155

                # Gradients of Theta1 based on training instance 1:
                #         -0.01270  -0.00165
                #         -0.01548  -0.00201
gradd1_1 = gradientD(weightlist1,delta1_1,a1)
print(gradd1_1)
print('\n')

print('example 1 instance 2 theta value')
delta1_2 = delta(weightlist1,a2,exceptout2,r2)
        # Computing gradients based on training instance 2
        #         delta3: [0.56597]
        #         delta2: [0.06740   0.08184]
print(delta1_2)
print('\n')

print('example 1 instance 2 gradient value')
                # Gradients of Theta2 based on training instance 2:
                #         0.56597  0.34452  0.33666

                # Gradients of Theta1 based on training instance 2:
                #         0.06740  0.02831
                #         0.08184  0.03437
gradd1_2 = gradientD(weightlist1,delta1_2,a2)
print(gradd1_2)
print('\n')

def transposelistoflist(l):
    newlistoflist = []
    for i in range(len(l[0])):
        newlist = []
        for j in range(len(l)):
            newlist.append(l[j][i])
        newlistoflist.append(newlist)
    return newlistoflist

listofgradient = [gradd1_1,gradd1_2]
gradientP1 = [lambda1*t for t in weightlist1]
grad_D_transpose = transposelistoflist(listofgradient)
grad_D_sum = [np.sum(t,axis=0) for t in grad_D_transpose]
```

```
update_gradients = []
for i in range(len(grad_D_sum)):
    update_gradients.append((grad_D_sum[i] + gradientP1[i])*(1/
 ↪numberofinstance1))

print('example 1 update gradient value')
print(update_gradients)
        # The entire training set has been processes. Computing the average␣
 ↪(regularized) gradients:
        #         Final regularized gradients of Theta1:
        #                 0.02735  0.01333
        #                 0.03318  0.01618

        #         Final regularized gradients of Theta2:
        #                 0.23000  0.14037  0.13756
print('\n')
print('end of example 1')
print('\n')
print('\n')

print('start of example 2')
# Initial Theta1 (the weights of each neuron, including the bias weight, are␣
 ↪stored in the rows):
#         0.42000  0.15000  0.40000
#         0.72000  0.10000  0.54000
#         0.01000  0.19000  0.42000
#         0.30000  0.35000  0.68000

# Initial Theta2 (the weights of each neuron, including the bias weight, are␣
 ↪stored in the rows):
#         0.21000  0.67000  0.14000  0.96000  0.87000
#         0.87000  0.42000  0.20000  0.32000  0.89000
#         0.03000  0.56000  0.80000  0.69000  0.09000

# Initial Theta3 (the weights of each neuron, including the bias weight, are␣
 ↪stored in the rows):
#         0.04000  0.87000  0.42000  0.53000
#         0.17000  0.10000  0.95000  0.69000
e2theta1 = np.array([[0.42,0.15,0.4],[0.72,0.1,0.54],[0.01,0.19,0.42],[0.3,0.
 ↪35,0.68]])
e2theta2 = np.array([[0.21,0.67,0.14,0.96,0.87],[0.87,0.42,0.2,0.32,0.89],[0.
 ↪03,0.56,0.8,0.69,0.09]])
e2theta3 = np.array([[0.04,0.87,0.42,0.53],[0.17,0.1,0.95,0.69]])
e2weightlist = [e2theta1,e2theta2,e2theta3]

# Training set
```

```
#         Training instance 1
#                 x: [0.32000   0.68000]
#                 y: [0.75000   0.98000]
#         Training instance 2
#                 x: [0.83000   0.02000]
#                 y: [0.75000   0.28000]

e2input1 = np.array([0.32,0.68])
e2input2 = np.array([0.83,0.02])
e2exceptout1 = np.array([0.75,0.98])
e2exceptout2 = np.array([0.75,0.28])

e2input1 = np.append(1,e2input1)
e2input2 = np.append(1,e2input2)
e2lambda0 = 0.25

print('example 2 instance 1 forward propagation')
e2r1,e2j1,e2a1 = forwardtest(e2input1,e2weightlist,e2exceptout1)
        # Processing training instance 1
        # Forward propagating the input [0.32000   0.68000]
        #         a1: [1.00000   0.32000   0.68000]

        #         z2: [0.74000   1.11920   0.35640   0.87440]
        #         a2: [1.00000   0.67700   0.75384   0.58817   0.70566]

        #         z3: [1.94769   2.12136   1.48154]
        #         a3: [1.00000   0.87519   0.89296   0.81480]

        #         z4: [1.60831   1.66805]
        #         a4: [0.83318   0.84132]

        #         f(x): [0.83318   0.84132]
        # Predicted output for instance 1: [0.83318   0.84132]
        # Expected output for instance 1: [0.75000   0.98000]
        # Cost, J, associated with instance 1: 0.791
print('\n')

print('example 2 instance 2 forward propagation')
e2r2,e2j2,e2a2 = forwardtest(e2input2,e2weightlist,e2exceptout2)
        # Processing training instance 2
        # Forward propagating the input [0.83000   0.02000]
        #         a1: [1.00000   0.83000   0.02000]

        #         z2: [0.55250   0.81380   0.17610   0.60410]
        #         a2: [1.00000   0.63472   0.69292   0.54391   0.64659]

        #         z3: [1.81696   2.02468   1.37327]
```

```
        #         a3: [1.00000   0.86020   0.88336   0.79791]

        #         z4: [1.58228   1.64577]
        #         a4: [0.82953   0.83832]

        #         f(x): [0.82953   0.83832]
        # Predicted output for instance 2: [0.82953   0.83832]
        # Expected output for instance 2: [0.75000   0.28000]
        # Cost, J, associated with instance 2: 1.944
print('\n')

e2jlist = np.array([e2j1,e2j2])
e2numberofinstance = 2

def sumofweights(listofweights,bias=True): # computes the square of all weights␣
 ↪of the network and sum them up
    sum = 0
    for weight in listofweights:
        if bias:
            w = weight.copy()
            w[:, 0] = 0
            sum += np.sum(np.square(w))
        else:
            sum += np.sum(np.square(weight))
    return sum

def overallcost(jlist,n,weightl,lambda_reg):
    s = sumofweights(weightl,bias=1)*lambda_reg/(2*n)
    jsum = np.sum(jlist)
    return jsum/n + s

print('example 2 overall cost')
print(overallcost(e2jlist,e2numberofinstance,e2weightlist,e2lambda0))
# Final (regularized) cost, J, based on the complete training set: 1.90351
print('\n')

print('example 2 backpropagation')
print('\n')

print('example 2 instance 1 delta')
e2delta1 = delta(e2weightlist,e2a1,e2exceptout1,e2r1)
# Running backpropagation
#         Computing gradients based on training instance 1
#                 delta4: [0.08318   -0.13868]
#                 delta3: [0.00639   -0.00925   -0.00779]
#                 delta2: [-0.00087   -0.00133   -0.00053   -0.00070]
print(e2delta1)
```

```
print('\n')

print('example 2 instance 2 gradient')
                # Gradients of Theta3 based on training instance 1:
                #         0.08318  0.07280  0.07427  0.06777
                #         -0.13868  -0.12138  -0.12384  -0.11300

                # Gradients of Theta2 based on training instance 1:
                #         0.00639  0.00433  0.00482  0.00376  0.00451
                #         -0.00925  -0.00626  -0.00698  -0.00544  -0.00653
                #         -0.00779  -0.00527  -0.00587  -0.00458  -0.00550

                # Gradients of Theta1 based on training instance 1:
                #         -0.00087  -0.00028  -0.00059
                #         -0.00133  -0.00043  -0.00091
                #         -0.00053  -0.00017  -0.00036
                #         -0.00070  -0.00022  -0.00048
e2grad1 = gradientD(e2weightlist,e2delta1,e2a1)
print(e2grad1)
print('\n')

print('example 2 instance 2 delta')
e2delta2 = delta(e2weightlist,e2a2,e2exceptout2,e2r2)
        # Computing gradients based on training instance 2
        #         delta4: [0.07953   0.55832]
        #         delta3: [0.01503   0.05809   0.06892]
        #         delta2: [0.01694   0.01465   0.01999   0.01622]
print(e2delta2)
print('\n')

print('example 2 instance 2 gradient')
                # Gradients of Theta3 based on training instance 2:
                #         0.07953  0.06841  0.07025  0.06346
                #         0.55832  0.48027  0.49320  0.44549

                # Gradients of Theta2 based on training instance 2:
                #         0.01503  0.00954  0.01042  0.00818  0.00972
                #         0.05809  0.03687  0.04025  0.03160  0.03756
                #         0.06892  0.04374  0.04775  0.03748  0.04456

                # Gradients of Theta1 based on training instance 2:
                #         0.01694  0.01406  0.00034
                #         0.01465  0.01216  0.00029
                #         0.01999  0.01659  0.00040
                #         0.01622  0.01346  0.00032
e2grad2 = gradientD(e2weightlist,e2delta2,e2a2)
print(e2grad2)
```

```
print('\n')

e2listofgradient = [e2grad1,e2grad2]
gradientP2 = [e2lambda0*t for t in e2weightlist]
for singleP in gradientP2:
    singleP[:, 0] = 0
e2_grad_D_transpose = transposelistoflist(e2listofgradient)
e2_grad_D_sum = [np.sum(t,axis=0) for t in e2_grad_D_transpose]
e2_update_gradients = []
for i in range(len(grad_D_sum)):
    e2_update_gradients.append((e2_grad_D_sum[i] + gradientP2[i])*(1/
 ↪e2numberofinstance))

print('example 2 update gradients')
print(e2_update_gradients)
        # The entire training set has been processes. Computing the average␣
 ↪(regularized) gradients:
        #         Final regularized gradients of Theta1:
        #                 0.00804  0.02564  0.04987
        #                 0.00666  0.01837  0.06719
        #                 0.00973  0.03196  0.05252
        #                 0.00776  0.05037  0.08492

        #         Final regularized gradients of Theta2:
        #                 0.01071  0.09068  0.02512  0.12597  0.11586
        #                 0.02442  0.06780  0.04164  0.05308  0.12677
        #                 0.03056  0.08924  0.12094  0.10270  0.03078

        #         Final regularized gradients of Theta3:
        #                 0.08135  0.17935  0.12476  0.13186
        #                 0.20982  0.19195  0.30343  0.25249
print('\n')
print('end of example 2')
```

1.3.4 stratified.py

[ ]:
```
from utils import *
from run import *
from neuralnetwork import *
from evaluationmatrix import *

# In this file, I reused the stratified cross-validation method from the last␣
 ↪assignment.

# Stratified K-Fold method
def stratifiedkfold(data, categorydict, k = 10):
    classindex = list(categorydict.values()).index("class")
```

```python
    datacopy = np.copy(data).T
    classes = list(Counter(datacopy[classindex]).keys())
    nclass = len(classes) # number of classes
    listofclasses = []

    for oneclass in classes:
        index = [idx for idx, element in enumerate(datacopy[classindex]) if element == oneclass]
        oneclassdata = np.array(datacopy.T[index])
        np.random.shuffle(oneclassdata)
        listofclasses.append(oneclassdata)

    splitted = [np.array_split(i, k) for i in listofclasses]
    combined = []

    for j in range(k):
        ithterm = []
        for i in range(nclass):
            if len(ithterm) == 0:
                ithterm = splitted[i][j]
            else:
                ithterm = np.append(ithterm,splitted[i][j],0)
        combined.append(ithterm)

    return combined

def ohe_stratifiedkfold(ohed_data, categorydict, k = 10):
    ohed_c = np.copy(ohed_data)
    n = 0
    classindices = []
    for i in categorydict:
        if categorydict[i] == "class_numerical":
            classindices.append(n)
        n += 1
    # nclass = len(classindices)
    listofclasses = []

    for index in classindices:
        ohed_copy = np.copy(ohed_c)
        # delete data with value !=1 at index
        ohed_copy = np.delete(ohed_copy, np.where((ohed_copy[:,index] == 0)), axis=0)
        # shuffle data
        np.random.shuffle(ohed_copy)
        listofclasses.append(ohed_copy)

    splitted = [np.array_split(i, k) for i in listofclasses]
```

```
    combined = []

    for j in range(k):
        ithterm = []
        for i in range(len(classindices)):
            if len(ithterm) == 0:
                ithterm = splitted[i][j]
            else:
                ithterm = np.append(ithterm,splitted[i][j],0)
        combined.append(ithterm)

    return combined


# def kfoldcrossvalidneuralnetwork(raw_data, rawcategory, layerparameter, k =␣
 ↪10, minibatchk = 15, lambda_reg = 0.15, learning_rate = 0.01, epsilon_0 = 0.
 ↪00001, softstop = 6000, printq = False):
#     folded = stratifiedkfold(raw_data, rawcategory, k)
#     listofnd = []
#     accuracylist = []
#     listofjlist = []
#     for i in range(k):
#         if printq:
#             print('fold',i+1)
#         rawtestdataset = folded[i].copy()
#         rawfoldedcopy = folded.copy()
#         rawfoldedcopy.pop(i)
#         rawtraindataset = np.vstack(rawfoldedcopy)
#         ohe_traindata,ohe_category = onehotencoder(rawtraindataset,␣
 ↪rawcategory)
#         ohe_testdata = onehotencoder(rawtestdataset, rawcategory)[0]
#         n_ohe_train,minmax = normalizetrain(ohe_traindata, ohe_category)
#         n_ohe_test = normalizealltest(ohe_testdata, ohe_category, minmax)
#         finalweight, jlist = train_neural_network(n_ohe_train, ohe_category,␣
 ↪layerparameter, minibatchk, lambda_reg, learning_rate, epsilon_0, softstop,␣
 ↪printq)
#         predictvsexpect, singleaccuracy = predict_many_nn(n_ohe_test,␣
 ↪ohe_category, finalweight)
#         listofnd.append(predictvsexpect)
#         accuracylist.append(singleaccuracy)
#         listofjlist.append(jlist)
#     acc = np.mean(accuracylist)
#     return listofnd, acc, listofjlist

def kfoldcrossvalidneuralnetwork(raw_data, rawcategory, layerparameter, k = 10,␣
 ↪minibatchk = 15, lambda_reg = 0.15, learning_rate = 0.01, epsilon_0 = 0.
 ↪00001, softstop = 6000, printq = False):
```

```
    ohe_data,ohe_category = onehotencoder(raw_data, rawcategory)
    folded = ohe_stratifiedkfold(ohe_data, ohe_category, k)
    listofnd = []
    accuracylist = []
    listofjlist = []
    for i in range(k):
        print('fold',i+1,'training in progress')
        if printq:
            print('fold',i+1)
        ohe_test = folded[i].copy()
        ohe_copy = folded.copy()
        ohe_copy.pop(i)
        ohe_train = np.vstack(ohe_copy)
        n_ohe_train,minmax = normalizetrain(ohe_train, ohe_category)
        n_ohe_test = normalizealltest(ohe_test, ohe_category, minmax)
        finalweight, jlist = train_neural_network(n_ohe_train, ohe_category,␣
 ↪layerparameter, minibatchk, lambda_reg, learning_rate, epsilon_0, softstop,␣
 ↪printq)
        predictvsexpect, singleaccuracy = predict_many_nn(n_ohe_test,␣
 ↪ohe_category, finalweight)
        print('fold',i+1,'training completed, accuracy = ',singleaccuracy)
        listofnd.append(predictvsexpect)
        accuracylist.append(singleaccuracy)
        listofjlist.append(jlist)
    acc = np.mean(accuracylist)
    return listofnd, acc, listofjlist
```

1.3.5 run.py

[ ]:
```
from utils import *
from stratified import *
from neuralnetwork import *

def importhousedata():
    house = importfile('hw3_house_votes_84.csv', ',')
    housecategory = {}
    for i in house[0]:
        housecategory[i] = 'categorical'
    housecategory["class"] = 'class'
    housedata = np.array(house[1:]).astype(float)
    return housedata, housecategory

def importwinedata():
    wine = importfile('hw3_wine.csv', '\t')
    winecategory = {}
    for i in wine[0]:
```

```
        winecategory[i] = 'numerical'
    winecategory["# class"] = 'class'
    winedata = np.array(wine[1:]).astype(float)
    return winedata, winecategory

def importcancerdata():
    cancer = importfile('hw3_cancer.csv', '\t')
    cancercategory = {}
    for i in cancer[0]:
        cancercategory[i] = 'numerical'
    cancercategory["Class"] = 'class'
    cancerdata = np.array(cancer[1:]).astype(float)
    return cancerdata, cancercategory

def importcmcdata():
    cmc = importfile('cmc.data', ',')
    cmccategory = {"Wife's age":"numerical","Wife's education":"categorical",
    "Husband's education":"categorical","Number of children ever born":
 ↪"numerical",
    "Wife's religion":"binary","Wife's now working?":"binary",
    "Husband's occupation":"categorical","Standard-of-living index":
 ↪"categorical",
    "Media exposure":"binary","Contraceptive method used":"class"}
    cmcdata = np.array(cmc).astype(int)
    return cmcdata, cmccategory


if __name__=="__main__":
    housedata, housecategory = importhousedata()
    winedata, winecategory = importwinedata()
    cancerdata, cancercategory = importcancerdata()
    cmcdata,cmccategory = importcmcdata()
    hiddenlayerparameter = [3,2]
    listoflistofoutputs, acc, listofjlist =␣
 ↪kfoldcrossvalidneuralnetwork(housedata, housecategory, hiddenlayerparameter,␣
 ↪k = 5, minibatchk = 3, lambda_reg = 0.1, learning_rate = 0.1, epsilon_0 = 0.
 ↪0001, softstop = 2000, printq = False)
    print(acc)
```

1.3.6 evaluationmatrix.py

[ ]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import display, Markdown

# I reused the code from my last assignment.
```

```
def accuracy(truePosi, trueNega, falsePosi, falseNega): # Count of all four
        return (truePosi+trueNega)/(truePosi+trueNega+falseNega+falsePosi)

def precision(truePosi, trueNega, falsePosi, falseNega):
        if (truePosi+falsePosi) == 0:
                return 0
        preposi = truePosi/(truePosi+falsePosi)
        # prenega = trueNega/(trueNega+falseNega)
        return preposi

def recall(truePosi, trueNega, falsePosi, falseNega):
        if (truePosi+falseNega)== 0:
                return 0
        recposi = truePosi/(truePosi+falseNega)
        # recnega = trueNega/(trueNega+falsePosi)
        return recposi

def fscore(truePosi, trueNega, falsePosi, falseNega, beta: 1):
        pre = precision(truePosi, trueNega, falsePosi, falseNega)
        rec = recall(truePosi, trueNega, falsePosi, falseNega)
        if (pre*(beta**2)+rec) == 0:
                return 0
        f = (1+beta**2)*((pre*rec)/(pre*(beta**2)+rec))
        return f

def evaluate(listsofoutput, positivelabel, beta=1):
    # list is list of [predicted, actual]
    listoftptnfpfn = []
    accuarcylists = []
    precisionlists = []
    recalllists = []
    fscorelists = []
    for output in listsofoutput:
        tp, tn, fp, fn, = 0, 0, 0, 0
        for i in range(len(output)):
            if output[i][0] == positivelabel and output[i][1] == positivelabel:
                tp += 1
            elif output[i][0] != positivelabel and output[i][0] == output[i][1]:
                tn += 1
            elif output[i][0] == positivelabel and output[i][1] != positivelabel:
                fp += 1
            elif output[i][0] != positivelabel and output[i][1] == positivelabel:
                fn += 1
        tptnfpfn = [tp, tn, fp, fn]
```

```
        listoftptnfpfn.append(tptnfpfn)
        accuarcylists.append(accuracy(tp, tn, fp, fn))
        precisionlists.append(precision(tp, tn, fp, fn))
        recalllists.append(recall(tp, tn, fp, fn))
        fscorelists.append(fscore(tp, tn, fp, fn, beta))
    return accuarcylists, precisionlists, recalllists, fscorelists,␣
 ↪listoftptnfpfn

def meanevaluation(listsofoutput, positivelabel, beta=1):
    accuarcylists, precisionlists, recalllists, fscorelists, notused =␣
 ↪evaluate(listsofoutput, positivelabel, beta)
    return sum(accuarcylists)/len(accuarcylists), sum(precisionlists)/
 ↪len(precisionlists), sum(recalllists)/len(recalllists), sum(fscorelists)/
 ↪len(fscorelists)

def markdownaprf(acc,pre,rec,fsc,beta,nvalue,title):
    acc, pre, rec, fsc = round(acc,3), round(pre,3), round(rec,3), round(fsc,3)
    display(Markdown(rf"""
        Result/Stat of {nvalue} trees random forest of {title}:
    | **Accuracy** | **Precision** | **Recall** | **F-Score, Beta={beta}** |
    | :---: | :---: | :---: | :---: |
    |{acc} | {pre} | {rec} | {fsc} |
    """))

def markdownmatrix(tptnfpfn,title):
    tp, tn, fp, fn = tptnfpfn[0], tptnfpfn[1], tptnfpfn[2], tptnfpfn[3]
    display(Markdown(rf"""
    Confusion Matrix: {title}
    |  | **Predicted +** | **Predicted-** |
    | :--- | :--- | :--- |
    | **Actual +** | {tp} | {fp} |
    | **Actual -** | {fn} | {tn} |
    """))

def confusionmatrix(truePosi, trueNega, falsePosi, falseNega, title=""):
        fig = plt.figure()
        plt.title(title)
        col_labels = ['Predict:+', 'Predict:-']
        row_labels = ['Real:+', 'Real:-']
        table_vals = [[truePosi, falseNega], [falsePosi, trueNega]]
        the_table = plt.table(cellText=table_vals,
                      colWidths=[0.1] * 3,
                      rowLabels=row_labels,
                      colLabels=col_labels,
                      loc='center')
        the_table.auto_set_font_size(False)
        the_table.set_fontsize(24)
```

```
    the_table.scale(4, 4)
    plt.tick_params(axis='x', which='both', bottom=False, top=False,␣
↪labelbottom=False)
    plt.tick_params(axis='y', which='both', right=False, left=False,␣
↪labelleft=False)

    for pos in ['right','top','bottom','left']:
        plt.gca().spines[pos].set_visible(False)

    plt.show()
    return
```