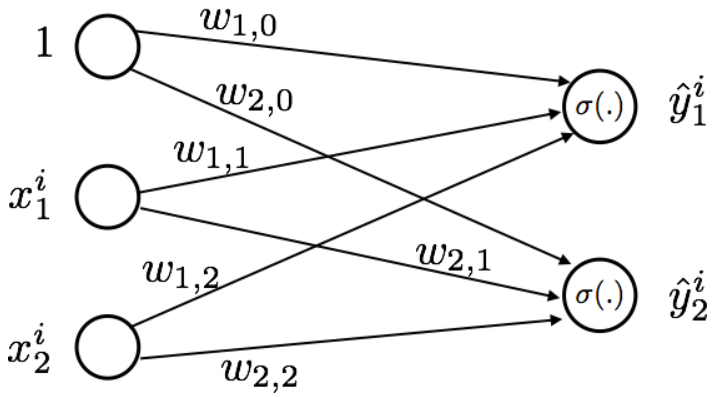
CS/ECE/ME532 Activity 24

Estimated Time: 25 minutes for P1, 25 minutes for P2

1. A script is available to train two neurons using stochastic gradient descent to solve two different classification problems. The two classifier structures are shown below. Here we use a logistic activation function $\sigma(z) = (1 + e^{-z})^{-1}$. The code generates



training data and labels corresponding to two decision boundaries: $x_2^i = -2x_1^i + 0.2$, and $x_2^i = 5(x_1^i)^3$.

a) Do you expect that a single neuron will be able to accurately classify data from case 1? Why or why not? Explain the impact of the bias term associated with $w_{1,0}$.

b) Do you expect that a single neuron will be able to accurately classify data from case 2? Why or why not? Explain the impact of the bias term associated with $w_{2,0}$.

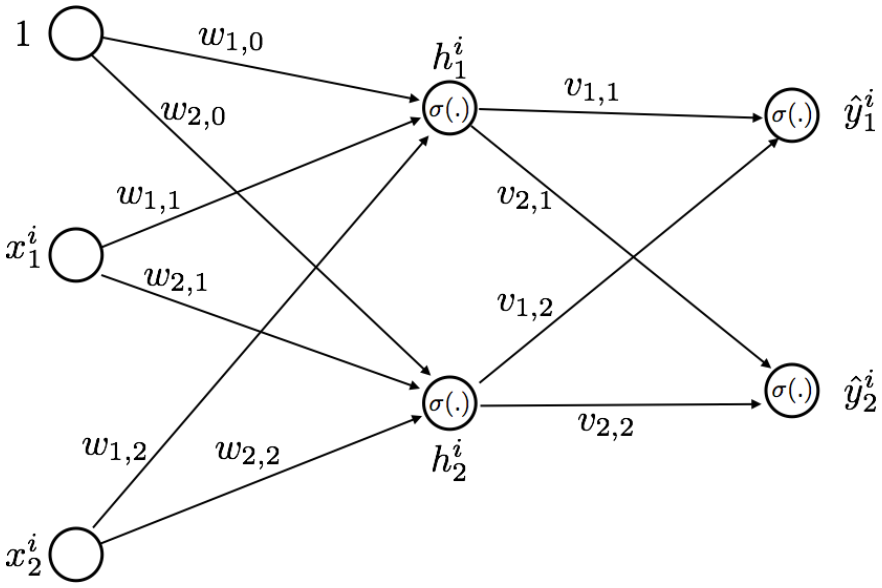
c) Run SGD for one epoch. This means you cycle through all the training data one time, in random order. Repeat this five times and find the average number of errors in cases 1 and 2.

d) Run SGD over twenty epochs. This means you cycle through all the training data twenty times, in random order. Repeat this five times and find the average number of errors in cases 1 and 2.

e) Explain the differences in classification performance for the two cases that result with both one and twenty epochs.

2. This remainder of this activity uses a three-layer neural network with three input nodes and two output nodes to solve two classification problems. We will vary the number

of hidden nodes. The figure below depicts the structure when there are two hidden nodes.



A second script is available that generates training data and trains the network using SGD assuming a logistic activation function $\sigma(z) = (1 + e^{-z})^{-1}$.

a) Use $M = 2$ hidden nodes and ten epochs in SGD. Run this four or five times and comment on the performance of the two classifiers and whether it varies from run to run.

b) Repeat $M = 2$ but use 100 epochs in SGD. (You may use fewer epochs if it takes more than a minute or two per run.) Run this several times and comment on the performance of the classifiers and whether it varies from run to run.

c) Recall the two-layer network results from the previous problem. How do the possible decision boundaries change when you add a hidden layer?

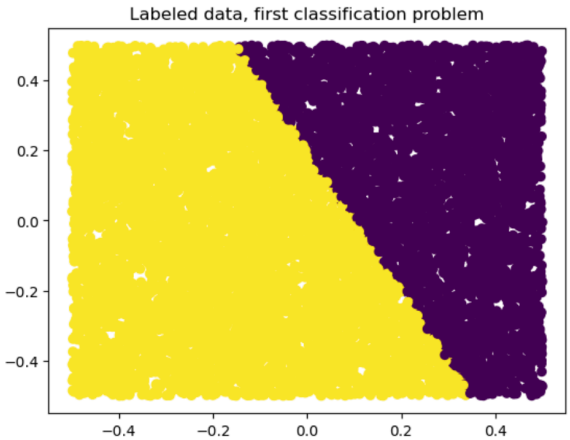
d) Now use $M = 3$ hidden nodes and run 100 epochs of SGD (or as many as you can compute). Does going from two to three hidden nodes affect classifier performance?

e) Repeat the previous part for $M = 4$ hidden nodes and comment on classifier performance.

532 Activity 24 DEVIN BRESSER

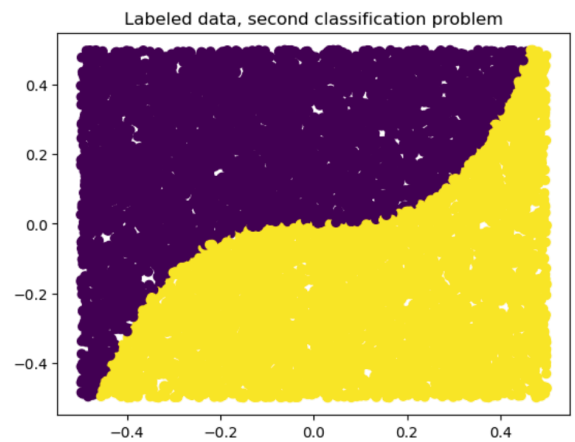
```
# Plot training data for first classification problem
plt.scatter(X[:,0], X[:,1], c=Y1.flatten())
plt.title('Labeled data, first classification problem')
plt.show()
```

1a




```
# Problem 1a comment: Yes, a single neuron should be able to
# classify this data well because it appears to be linearly separable.
# The bias term w_1,0 is crucial for this to work though.
# It represents the y-intercept of the decision boundary.
```

1b




```
In [4]: # Problem 1b comment: No, a single neuron will not be able to
        # accurately classify this data. A single neuron can only learn to classify
        # data linearly (straight line decision boundary).
        # The bias term w_2,0 in this case would add a y-intercept to the linear
        # decision boundary, so we would at least get the best possible line.
```

1c

[48]:
```
## Train NN
Xb = np.hstack((np.ones((n,1)), X))
q = np.shape(Y)[1] #number of classification problems
M = 3 #number of hidden nodes

## initial weights
W = np.random.randn(p+1, q);

alpha = 0.1 #step size
L = 1 #number of epochs

R = 5 # num trials

def logsig(_x):
    return 1/(1+np.exp(-_x))

classifier_1_num_errors = []
classifier_2_num_errors = []

for a in range(R):

    for epoch in range(L):
        ind = np.random.permutation(n)
        for i in ind:
            # Forward-propagate
            Yhat = logsig(Xb[[i],:]@W)
             # Backpropagate
            delta = (Yhat-Y[[i],:])*Yhat*(1-Yhat)
            Wnew = W - alpha*Xb[[i],:].T@delta
            W = Wnew
        #print('epoch: ', epoch)

    ## Final predicted labels (on training data)
    H = logsig(np.hstack((np.ones((n,1)), Xb@W)))
    Yhat = logsig(Xb@W)

    err_c1 = np.sum(abs(np.round(Yhat[:,0])-Y[:,0]))
    #print('Errors, first classification problem:', err_c1)

    err_c2 = np.sum(abs(np.round(Yhat[:,1])-Y[:,1]))
    #print('Errors, second classification problem:', err_c2)

    classifier_1_num_errors.append(err_c1)
    classifier_2_num_errors.append(err_c2)

print(f"Mean # errors, classifier 1, {R} trials of {L} epochs: {np.mean(classifier_1_num_err
print(f"Mean # errors, classifier 2, {R} trials of {L} epochs: {np.mean(classifier_2_num_err
```

```
Mean # errors, classifier 1, 5 trials of 1 epochs: 125.6
Mean # errors, classifier 2, 5 trials of 1 epochs: 743.8
```

1d

[49]:
```
## Train NN
Xb = np.hstack((np.ones((n,1)), X))
q = np.shape(Y)[1] #number of classification problems
M = 3 #number of hidden nodes

## initial weights
W = np.random.randn(p+1, q);

alpha = 0.1 #step size
L = 20 #number of epochs

R = 5 # num trials

def logsig(_x):
    return 1/(1+np.exp(-_x))

classifier_1_num_errors = []
classifier_2_num_errors = []

for a in range(R):

    for epoch in range(L):
        ind = np.random.permutation(n)
        for i in ind:
            # Forward-propagate
            Yhat = logsig(Xb[[i],:]@W)
             # Backpropagate
            delta = (Yhat-Y[[i],:])*Yhat*(1-Yhat)
            Wnew = W - alpha*Xb[[i],:].T@delta
            W = Wnew
        #print('epoch: ', epoch)

    ## Final predicted labels (on training data)
    H = logsig(np.hstack((np.ones((n,1)), Xb@W)))
    Yhat = logsig(Xb@W)

    err_c1 = np.sum(abs(np.round(Yhat[:,0])-Y[:,0]))
    #print('Errors, first classification problem:', err_c1)

    err_c2 = np.sum(abs(np.round(Yhat[:,1])-Y[:,1]))
    #print('Errors, second classification problem:', err_c2)

    classifier_1_num_errors.append(err_c1)
    classifier_2_num_errors.append(err_c2)

print(f"Mean # errors, classifier 1, {R} trials of {L} epochs: {np.mean(classifier_1_num_err
print(f"Mean # errors, classifier 2, {R} trials of {L} epochs: {np.mean(classifier_2_num_err
```

```
Mean # errors, classifier 1, 5 trials of 20 epochs: 32.8
Mean # errors, classifier 2, 5 trials of 20 epochs: 749.4
```

1e

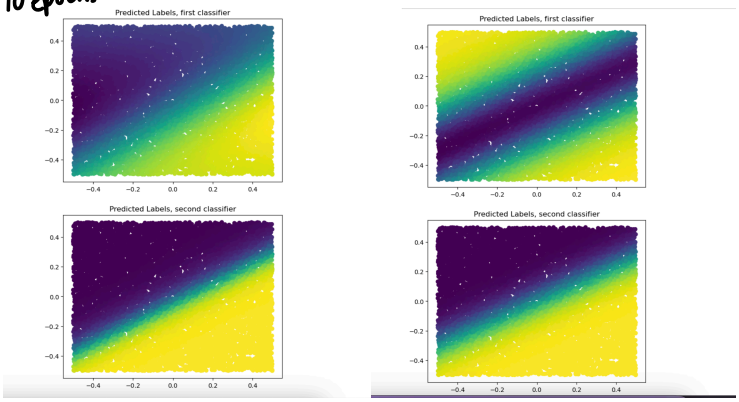
Comment: This result supports my earlier comments that the first dataset can be well classified by a single neuron given enough training (# epochs). On the second dataset, we observe that the number of training epochs does not improve the classifier at all because a single neuron is fundamentally unsuited for the task.

2a The first classifier varies from run to run.

It is not classifying the first dataset well.

The second one does not vary. It classifies the second dataset poorly (linear boundary for curvy data).

10 epochs



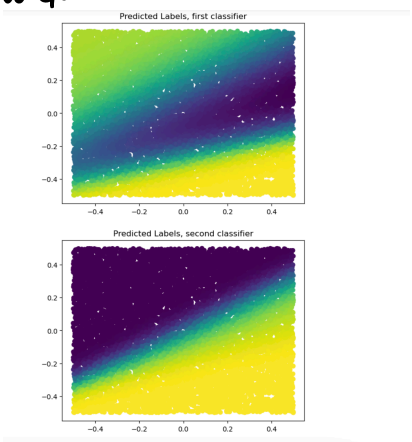
2b

The results are consistent now.

They look like this every time →

The classification results are still very poor.

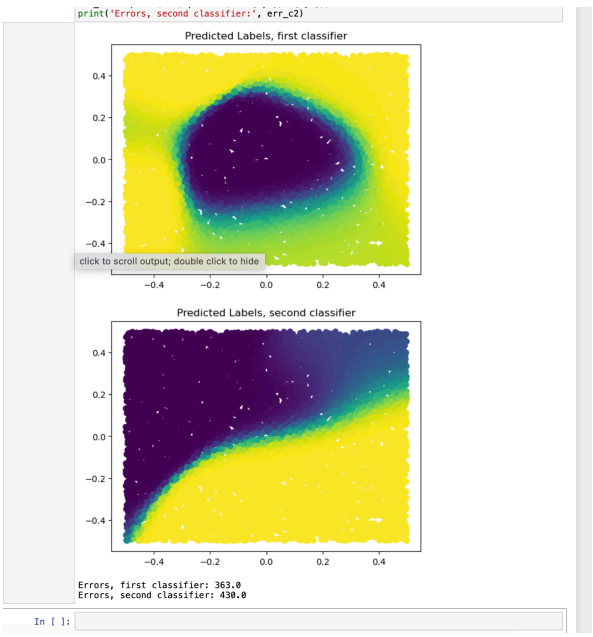
100 epochs



2c A network with a hidden layer can create nonlinear decision boundaries.

2d

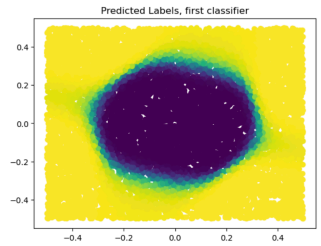
Yes, classifier performance is massively improved.

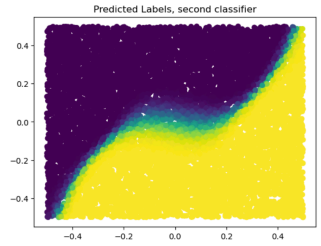


2e

Adding a fourth hidden node improves the classifiers even more.

```
err_c2 = np.sum(abs(np.round(Yhat[:,1])-Y[:,1]))
print('Errors, second classifier:', err_c2)
```






```
Errors, first classifier: 137.0
Errors, second classifier: 79.0
```

In [ ]: